Mathematical Methods for Non-Intrusive Load Monitoring

by

Zachary Remscrim

S.B., E.E.C.S., S.B., Mathematics, M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2010

[June 2010]

Author _____

Department of Electrical Engineering and Computer Science

May 17, 2010

Certified by _____

Dr. Steven B. Leeb

Thesis Supervisor

Accepted by _____

Dr. Christopher J. Terman

Chairman, Department Committee on Graduate Theses

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
JUN 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Mathematical Methods for Non-Intrusive Load Monitoring	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Massachusetts Institute of Technology,77 Massachusetts Avenue,Cambridge,MA,02139**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The calculation of the Discrete Fourier Transform (DFT) of a discrete time signal is a fundamental problem in discrete-time signal processing. This thesis presents algorithms that use methods from number theory and algebra to exploit additional constraints about a signal to aid in the calculation of its DFT. First, an algorithm is presented that estimates the DFT of an unquantized signal given only a quantized version of that signal. Second, an algorithm to estimate the value of one subset of DFT coefficients from knowledge of another subset of DFT coefficients, for an appropriately constrained class of waveforms, is presented and analyzed. Thirdly, an algorithm to classify electrical loads on the basis of a subset of the DFT coefficients of load current is demonstrated. Finally an embedded system that calculates DFT coefficients of measured current and makes this information available in convenient forms is considered.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**197**	

The Theory and Application of Non-Intrusive Load Monitoring
by
Zachary Remscrim

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

The calculation of the Discrete Fourier Transform (DFT) of a discrete time signal is a fundamental problem in discrete-time signal processing. This thesis presents algorithms that use methods from number theory and algebra to exploit additional constraints about a signal to aid in the calculation of its DFT. First, an algorithm is presented that estimates the DFT of an unquantized signal given only a quantized version of that signal. Second, an algorithm to estimate the value of one subset of DFT coefficients from knowledge of another subset of DFT coefficients, for an appropriately constrained class of waveforms, is presented and analyzed. Thirdly, an algorithm to classify electrical loads on the basis of a subset of the DFT coefficients of load current is demonstrated. Finally, an embedded system that calculates DFT coefficients of measured current and makes this information available in convenient forms is considered.

Thesis Supervisor: Dr. Steven B. Leeb
Title: Professor, Laboratory for Electromagnetic and Electronic Systems

Acknowledgements

I would like to thank Professor Steven Leeb for his continuous help and guidance in all my research endeavors, Warit Wichakool for his collaboration on the cross estimation problem, and James Paris for many useful discussions. This research was funded by the Grainger Foundation, the BP-MIT Alliance, the Office of Naval Research under the ESRDC program, the MIT Sea Grant College Program, and by the MIT Center for Materials Science and Engineering. The author also gratefully acknowledge the advice and support of Dr. Manny Landsman.

Contents

Chapter 1

Introduction

This thesis presents and analyzes algorithms that solve a variety of long standing problems in non-intrusive power system monitoring. Techniques from number theory and algebra are applied to solve common power system monitoring problems, such as accurately determining the harmonic content of the current drawn by an electrical load given only a coarsely quantized version of that current and classifying an unknown load on the basis of the current drawn by that load. The methods of this thesis can be applied to a variety of other common discrete-time signal processing tasks that involve computation of the Discrete Fourier Transform (DFT) of a signal.

Conventional sub-metering of individual electrical loads to detect problems and conduct energy score-keeping has long been costly and inconvenient. A nagging problem for over two decades has been that these costs increase swiftly as data requirements become increasingly complex: "the high cost of equipment continues to limit the amount of [usage] data utilities can collect. Additional drawbacks of the equipment now available for collection of end-use load survey data range from their cost, reliability, and flexibility to intrusion into the customer's activities and premises" [19].

Computational power and data transmission capabilities for commercial monitoring and control systems have out-paced the problem of putting sensors in all the right places. Various kinds of high-speed data networks provide convenient remote access to control inputs and system operating information for embedded control and monitoring systems. Similarly, microprocessors and associated technologies for these systems have achieved astounding price/performance ratios. Obtaining useful information, however, generally requires proper installation, maintenance, and interpretation of a vast collection of sensors a daunting proposition even if the sensors are mass produced, micro-miniature, and individually inexpensive.

A Non-Intrusive Load Monitor (NILM) can determine the electrical operating schedule of a collection of loads from a single measurement of aggregate current flowing to the loads. The NILM addresses the "sensor problem" for electric load monitoring by extracting information about individual loads from limited measurements at an easy-to-access, centralized location [1]. For example, the NILM can disaggregate and report the operation of individual electrical loads like lights and motors from measurements made only at the electric meter where service is provided to a building. The NILM is capable of performing this disaggregation even when many loads are operating at the same time. Because the NILM associates observed electrical waveforms with individual kinds of loads, it is possible to exploit modern state and parameter estimation algorithms to remotely verify and determine the condition or health of critical loads ([20] describes techniques suitable for motor parameter estimation from a non-intrusive monitor, for example.). The NILM has the potential to be a turn-key, enabling platform for future energy conservation and monitoring in a smart grid that services both homes and commercial/industrial facilities.

A NILM makes us of the "spectral envelope" representation of observed current

signals. This scheme considers samples $i[n]$ of a current $i(t)$, where a set of samples are taken for each period of the line voltage waveform. The DFT of the set of samples corresponding to each period is then computed. This produces a set of DFT coefficients for each period. A spectral envelope is the time evolution of a single DFT coefficient. This can be a very flexible basis for computing and tracking all sorts of useful metrics about power consumption. Spectral envelopes estimate real and reactive power consumption and harmonic content. The algorithms presented in this thesis can be applied to a variety of useful spectral envelope calculations.

When working with a continuous-time signal $i(t)$, it is often desirable to examine its discrete-time samples $i[n]$. In any practical application, it is impossible to obtain $i[n]$ to infinite resolution. Instead, only the quantized values $\bar{i}[n]$ are available, where $\bar{i}[n]$ is simply $i[n]$ quantized to some finite number of bits of resolution. While the DFT of this quantized signal can be redily computed, this is not a perfectly accurate statement of the true frequency content of the unquantized signal $i[n]$. Unfortunately, since quantization is a many-to-one operation, it is, in general, impossible to exactly reconstruct $i[n]$ from $\bar{i}[n]$, and thus it is also impossible to exactly determine the DFT of $i[n]$ from $\bar{i}[n]$, because the DFT is a bijection. However, with additional information about the structure of $i[n]$, it is possible to obtain a significantly more accurate estimate of the true frequency content of $i[n]$. Additional constraints about $i[n]$ restrict the class of possible $i[n]$ that could have produced the observed $\bar{i}[n]$. Chapter 2 demonstrates an efficient algorithm that exploits the structure of the mapping between regions of frequency space and quantized values to improve the estimation of the frequency content of $i[n]$ by applying these additional constraints.

Chapter 3 considers the related question of using knowledge of the values of some particular subset of frequency components to estimate the values of other frequency

components. Of course, due to the orthogonality of frequency components, the value of one component is complete independent from the value of any other component, so no non-trivial answer can be given to this question, in general. However, if certain additional constraints about $i[n]$ are known, then it will be possible to use information about one set of frequency components to estimate the value of another set. The types of constraints that make this possible are considered. An initial algorithm is presented that applies those constraints to perform this estimation. This algorithm will be shown to be numerically unstable, and a refined algorithm will be considered that completely avoids numerical problems by using properties of cyclotomic fields.

In Chapter 4, the problem of identifying an electrical load from a subset of the frequency content of its measured current is considered. The relationship between the structure of the currents drawn by different loads in a class and the minimal subset of frequency content needed to unambiguously identify a single load in that class is examined. An algorithm for performing this classification task is discussed.

Chapter 5 details the design and operation of an implementation of an embedded system that takes quantized samples $\bar{i}[n]$ of a current signal and computes the corresponding spectral envelopes. The system is capable of delivering this information in a variety of convenient forms, including via WiFi.

Chapter 2

Quantization Effects on the DFT

2.1 Spectral Envelopes

The spectral envelopes of current represent the harmonic content of the input waveform for each line-locked period of the service voltage. Given N samples $i[n]$ of a waveform $i(t)$ over one period, the samples can be expressed in terms of their spectral content by

$$i[n] = \frac{1}{N} \sum_{k=0}^{N-1} \left(p_k \sin\left(\frac{2\pi kn}{N}\right) + q_k \cos\left(\frac{2\pi kn}{N}\right) \right) \tag{2.1}$$

where the spectral envelope values p_k and q_k for that period are defined as

$$p_k = \sum_{n=0}^{N-1} i[n] \sin\left(\frac{2\pi kn}{N}\right) \tag{2.2}$$

and

$$q_k = \sum_{n=0}^{N-1} i[n] \cos\left(\frac{2\pi kn}{N}\right). \tag{2.3}$$

Here, k denotes the multiple of the line frequency to which a particular spectral envelope corresponds; for example, on a 60Hz utility service, $k = 1$ corresponds to the 60 Hz component and $k = 3$ to the 180 Hz component. The values of these spectral envelopes are calculated for each period of the line voltage; the values at period m will be denoted $p_k[m]$ and $q_k[m]$. With this definition, spectral envelopes can naturally be calculated from the real and imaginary parts of the Discrete Fourier Transform (DFT) [2] of $i[n]$ over each period of the line voltage.

The complete collection of coefficients p_k and q_k, for all k, determine the signal $i[n]$ over one cycle. The spectral envelope values can be understood to have meaningful physical interpretations. For example, if the line voltage waveform consists of only a single pure sinusoid, then p_1 corresponds to the real power consumed, and q_1 to the reactive power.

2.2 Quantization

In any practical application, it is not possible to obtain samples $i[n]$ of the waveform $i(t)$ to infinite precision. Instead, only quantized samples are generally available. A quantizer maps points in a continuous interval to a discrete set of points. The continuous interval is partitioned into a set of regions, called quantization intervals, by a set of points, called boundary points or interval endpoints. Each interval has a value associated with it; these values are called representation points. The quantizer maps each value in a quantization interval to the corresponding representation point. To formally specify the operation of a quantizer, let $M \in \mathbb{N}$ and define two sets of points $A = \{a_1, \ldots, a_M\}$ and $B = \{b_0, \ldots, b_M\}$ where $a_j, b_j \in \mathbb{R}\ \forall j$ and $b_j < b_{j+1}\ \forall j \in [0, M-1]$. The elements of the set A are the representation points and the elements of set B are the boundary points.

Set B specifies a collection of M regions, $R_1, \ldots, R_M$ where $R_1 = (b_0, b_1]$, $R_2 = (b_1, b_2]$, $R_3 = (b_2, b_3]$, $\ldots$, $R_M = (b_{M-1}, b_M]$. Define the quantizer function, $Q : (b_0, b_M] \rightarrow B$, such that $Q(x) = b_j$ where $x \in R_j$. In the above definition, we restrict the domain of the quantizer to the interval $(b_0, b_M]$. The operation of the quantizer is left undefined for inputs outside of all quantization intervals. An alternate approach would be to define the first and last quantization intervals to be $R_1 = (-\infty, b_1]$ and $R_M = (b_{M-1}, \infty]$, which would have the advantage of having the entire real line as a domain. However, this definition is not used here because unbounded quantization intervals would unnecessarily complicate the analysis without providing any benefit because any practical application involves quantizing values in a finite interval.

Let $\bar{i}[n]$ denote the quantized samples of the waveform $i(t)$. That is to say, $\bar{i}[n] = Q(i[n])$. In an analogous fashion to the above, these quantized samples can be expressed in terms of their spectral content by

$$\bar{i}[n] = \frac{1}{N} \sum_{k=0}^{N-1} \left(\overline{p}_k \sin\left(\frac{2\pi kn}{N}\right) + \overline{q}_k \cos\left(\frac{2\pi kn}{N}\right) \right) \tag{2.4}$$

where the spectral envelopes $\overline{p}_k$ and $\overline{q}_k$ for that period are defined by

$$\overline{p}_k = \sum_{n=0}^{N-1} \bar{i}[n] \sin\left(\frac{2\pi kn}{N}\right) \tag{2.5}$$

and

$$\overline{q}_k = \sum_{n=0}^{N-1} \bar{i}[n] \cos\left(\frac{2\pi kn}{N}\right). \tag{2.6}$$

It is useful to consider the relationship between the p_k's and q_k's that would be desirable to have and the $\overline{p}_k$'s and $\overline{q}_k$'s that one be obtained in practice. To this end,

first notice that there is a one-to-one relationship between the set of all p_k's and q_k's and the N samples $i[n]$ (because the DFT is a bijection). Similarly, there is a one-to-one relationship between the set of all $\overline{p}_k$'s and $\overline{q}_k$'s and the N quantized samples $\overline{i}[n]$. On the other hand, there is a many-to-one relationship between the N unquantized samples $i[n]$ and the N quantized samples $\overline{i}[n]$ (the quantizer maps many unquantized values to the same quantized value, because all points in a quantization interval are mapped to the same representation point). Thus, there is a many-to-one relationship between the set of all p_k's and q_k's and the set of all $\overline{p}_k$'s and $\overline{q}_k$'s, which means it is impossible to uniquely reconstruct the p_k's and q_k's from the $\overline{p}_k$'s and $\overline{q}_k$'s.

Despite this non-uniqueness problem, one can still consider how accurately the p_k's and q_k's can be estimated from the $\overline{p}_k$'s and $\overline{q}_k$'s. One simple method is to use each $\overline{p}_k$ as the estimate for the corresponding p_k and each $\overline{q}_k$ as an estimate for the $\overline{q}_k$. Fig. 2.1 shows $\overline{p}_1$ values as a function of actual p_1 values for a pure 60 Hz in-phase sinusoid of varying amplitudes, with $N = 128$ and 4-bit samples. Values of $\overline{p}_1$ are marked with a $+$. The line $\overline{p}_1 = p_1$ is included for reference to illustrate how close each actual $\overline{p}_1$ value is to the corresponding p_1 value. Clearly, using $\overline{p}_1$ as an estimate for p_1 is reasonably accurate, though there is noticable error.

It is possible to obtain better estimates for the p_k's and q_k's. As noted above, there is a one-to-one relationship between the set of all p_k's and q_k's and the N samples in time $i[n]$. For notational convenience, let the space of all possible values of the p_k's and q_k's be referred to as PQ-space. Clearly, PQ-space is isomorphic to $\mathbb{R}^N$ because each frequency component can have an arbitrary real value. Also, as noted above, many sets of N samples $i[n]$ map to the same set of N samples $\overline{i}[n]$; thus, many points in PQ-space map to the same set of N samples $\overline{i}[n]$. These points form a region in PQ-space. The following lemma states certain useful properties of these regions.

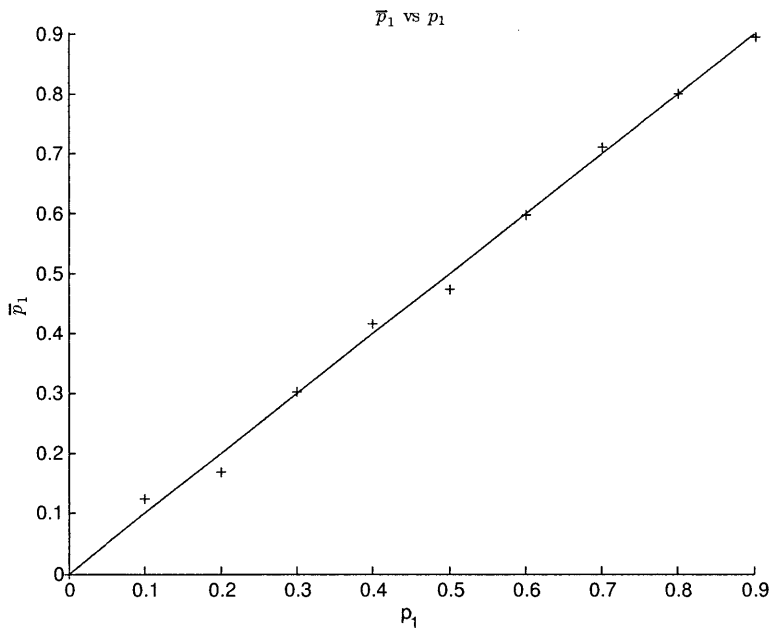


Figure 2.1: $\overline{p}_1$ vs p_1.

Lemma 1. *Let A be the set of points in PQ-space that corresponds to some arbitrary set of quantized samples $\overline{i}[n]$. Then A has the following properties:*

1. *A is connected.*
2. *A is convex.*
3. *A is a bounded polytope.*

Proof. 1. Assume, for purpose of contradiction, that A is not connected. By definition, this means that A can be expressed as the union of two non-empty open sets, as shown in Fig. 2.2. We can select three colinear points, x, y, z, as shown in the figure. Consider moving along the line segment from x to y to z. Each point in PQ-space has a corresponding unique set of unquantized samples, $i[n]$. Thus, as we move along this line segment, the corresponding set of unquantized samples change. To be precise, the direction specified by motion along the line segment corresponds to

a particular ratio of spectral components (the p_k's and q_k's). Moving in a given direction causes the unquantized samples to change by adding spectral content in the specified ratio. For example, if both x and y lied along the p_1 axis, then moving along the line segment causes the p_1 content of the samples to change, but all other spectral content is left unaffected. Similarly, if x and y lie in the p_1q_1 plane along the line $p_1 = 2q_1$ then moving along the line segment causes the p_1 and q_1 content to change (with the p_1 content changing by twice as much as the q_1 content), and all other spectral content to remain the same.

Every point in A corresponds to the same collection of quantized samples, $\bar{i}[n]$. This means that as we move along the line segment the first time an unquantized sample will cross a boundary point of any quantization interval is at the boundary of the set containing x and y. At this point, some sample, say $i[k]$, will leave the quantization interval whose representation point is $\bar{i}[k]$ and move to either the quantization interval immediately above or the one immediately below. In particular, if the added spectral content at sample index k (in the ratio specified by moving along the line segment from x to y) is positive (that is to say, if we consider the time domain samples corresponding to just the added spectral content from moving along the line segment, and the sample at index k is positive) then this sample moves to the quantization interval immediately above and if the added spectral content at sample index k is negative then this sample moves to the quantization interval immediately below. The added spectral content cannot be 0 at sample index k because, if it were, then motion along the line segment would not cause $i[k]$ to move at all, which contradicts the fact that $i[k]$ crossed a boundary point of a quantization interval. In any case, whichever direction $i[k]$ moved along the path from x to y, it must move in the same direction along the path from y to z

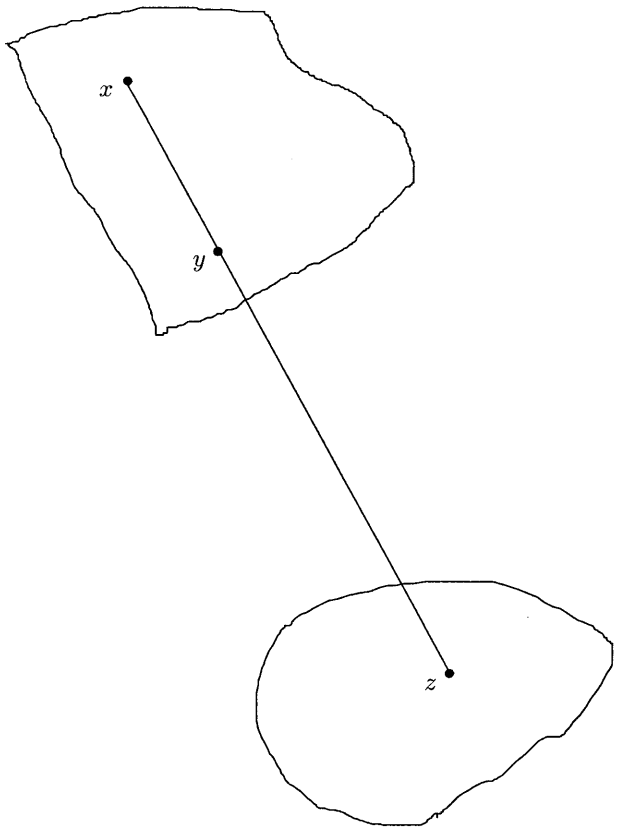


Figure 2.2: A is connected.

and so $i[k]$ can't return to the original quantization interval at z. Thus, $i[k]$ would be in different quantization intervals at x and z. This means that x and z must correspond to different sets of quantized samples, which contradicts the definition of A. This contradiction immediately implies that A is connected.

2. The fact that A is convex follows from an analogous argument. Assume, for contradiction, that A is not convex. Then we have the situation shown in Fig. 2.3. We can again select three colinear points, x, y, z, as labeled, and consider traveling along the straight line path specified. We again have the problem that once a sample point leaves a quantization interval, it can never return, and so x and z again correspond to different quantized samples. This contradicts the definition of A and so A must be convex.

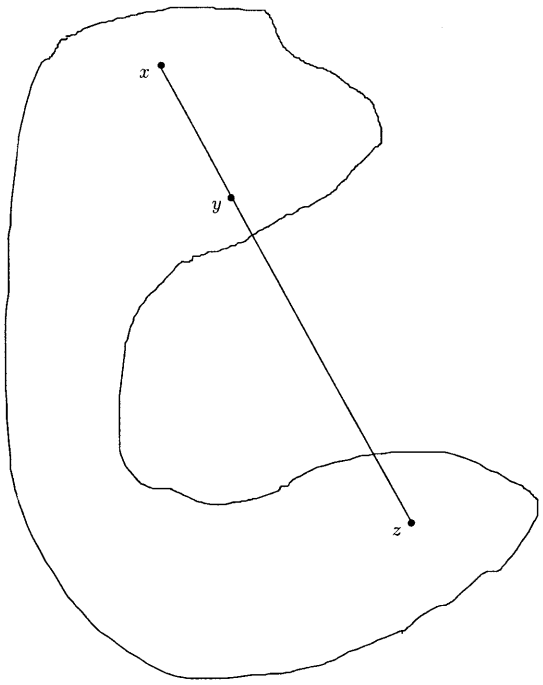


Figure 2.3: A is convex.

3. To see that A is a polytope, consider the boundary between A and another neighboring set of points B in PQ-space that corresponds to a different set of quantized samples. By the above, both A and B are convex, and so the boundary must be a portion of a hyperplane. The hyperplane divides PQ-space into two half-spaces where A only includes points from one of the half-spaces and B only includes points from the other half-space. Thus, A is the intersection of the half-spaces specified by all of the boundary hyperplanes. This intersection forms, by definition, a polytope. The polytope is necessarily bounded because the boundary points of the quantization region $\bar{i}[n]$ bound the range that $i[n]$ can be in while still remaining in A, $\forall n$. It is worth noting that PQ-space is, itself, unbounded (as noted above, it is isomorphic to $\mathbb{R}^N$), but the only polytopes of interest in PQ-space are those that correspond to quantized samples $\bar{i}[n]$, which are all bounded.

□

One possible way to improve the accuracy of the estimation of the p_k's and q_k's from the $\overline{p}_k$'s and $\overline{q}_k$'s is to use the fact that there is a one-to-one correspondence between quantized samples $\bar{i}[n]$ and regions in PQ-space. Moreover, as will be shown shortly, it is possible to determine the region from the quantized samples. Since it is also clearly possible to determine the quantized samples from the $\overline{p}_k$'s and $\overline{q}_k$ (using the DFT), the $\overline{p}_k$'s and $\overline{q}_k$'s can be used to determine the region in PQ-space that the true (unquantized) samples came from. Once this region is determined, it can be used to estimate the p_k's and q_k's in several ways. A particularly simple estimate would be to use the centroid of the region of PQ-space. Another technique involves the use of additional information about the behavior of real electrical loads. As observed in [1], many electrical loads draw current profiles that consist of only a small number of significantly non-zero spectral envelopes, for example the 1^{st},3^{rd},5^{th},and 7^{th} (in both p and q). If it is known that only a small number of the p_k's and q_k's are non-zero, this knowledge could be exploited by considering only the intersection of the region in PQ-space with the subspace spanned by the non-zero p_k's and q_k's and then taking the centroid of the intersection. This second estimation technique is considered here.

To better understand the structure of these regions of PQ-space, consider the following concrete example of a signal for which only p_1 and p_3 are nonzero. Let $i[n] = 0.51\sin(\frac{2\pi n}{N})+0.23\sin(\frac{6\pi n}{N})$ denote $N = 16$ 4-bit samples of a signal. In practice, of course, both the number of samples N and the bit resolution will generally be significantly higher than in this example; these values are chosen to give a simple illustration of the structure of PQ-space. Figure 2.4 shows the underlying waveform $i(t)$, the sample values $i[n]$, the quantized samples $\bar{i}[n]$ as well as the upper and lower bounds on the true value of each $i[n]$ given the observed $\bar{i}[n]$ (these are simply the upper and lower boundary points of the quantization interval that each $i[n]$ is mapped to). The signal $i[n]$ corresponds to a

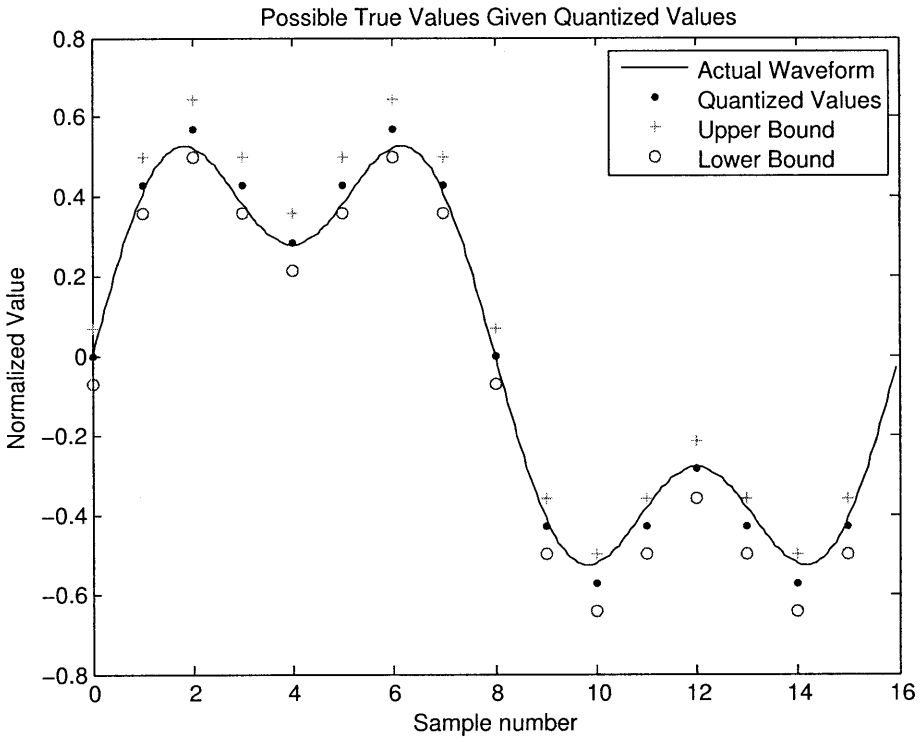


Figure 2.4: An example of quantization boundaries and representation points.

point in PQ-space, namely the point $(.51, .23)$ in the p_1p_3-plane. The quantized signal $\overline{i[n]}$ corresponds to a region in PQ-space. This region is depicted in Fig. 2.5. Finally, Fig. 2.6 shows a collection of regions of PQ-space in the neighborhood of the above point. As can be seen from these figures, the regions in PQ-space are highly non-uniform.

The following section presents an algorithm that determines the region in PQ-space corresponding to quantized samples $\bar{i}[n]$. The effectiveness of the technique is illustrated in Fig. 2.7. As was the case in the example shown in Fig. 2.1, a family of pure 60 Hz in-phase sinusoids of varying amplitudes, are sampled with $N = 128$ 4-bit samples. The estimates produced by the second estimation technique discussed above are marked with + symbols. Again, the reference line showing the true p_1 value is shown to illustrate the error. Clearly, this method produces more accurate estimates than simply using $\overline{p}_1$ as an estimate for p_1.

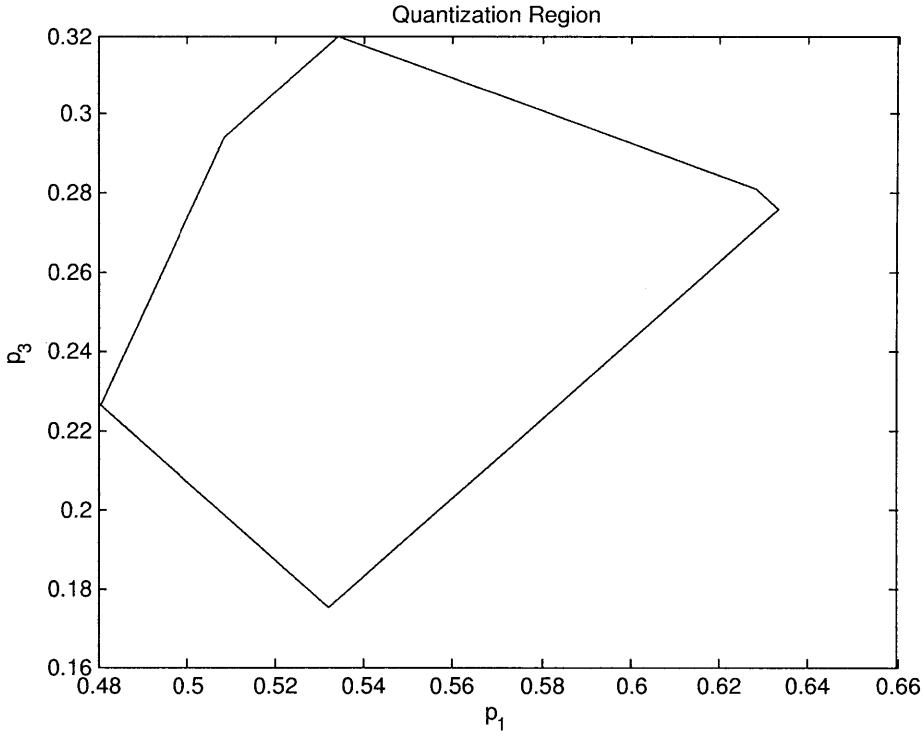


Figure 2.5: A single region in PQ-space.

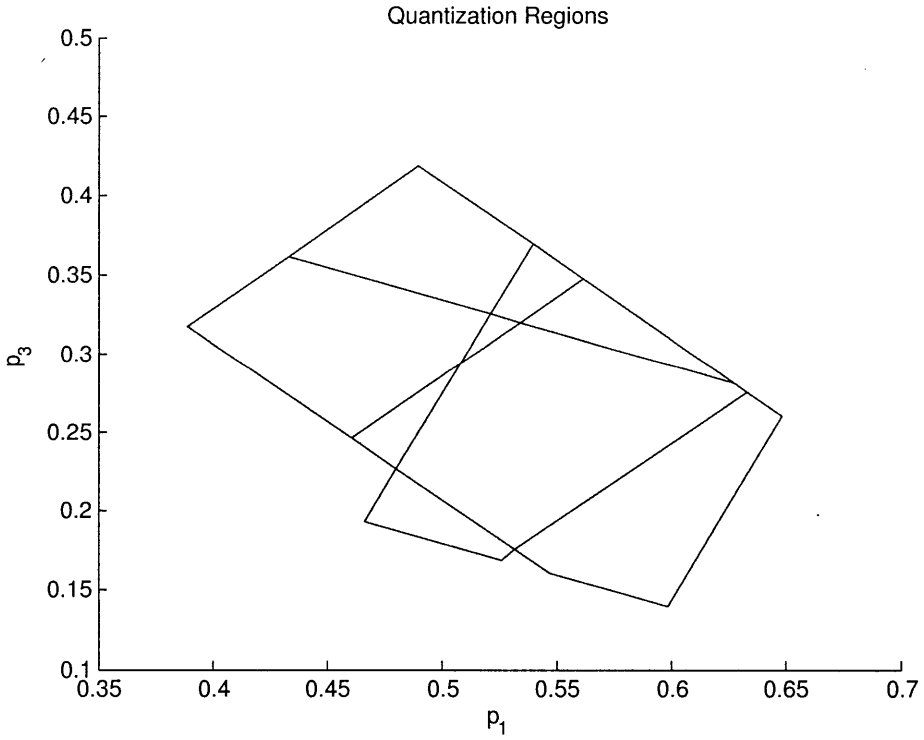


Figure 2.6: A set of neighboring regions in PQ-space.

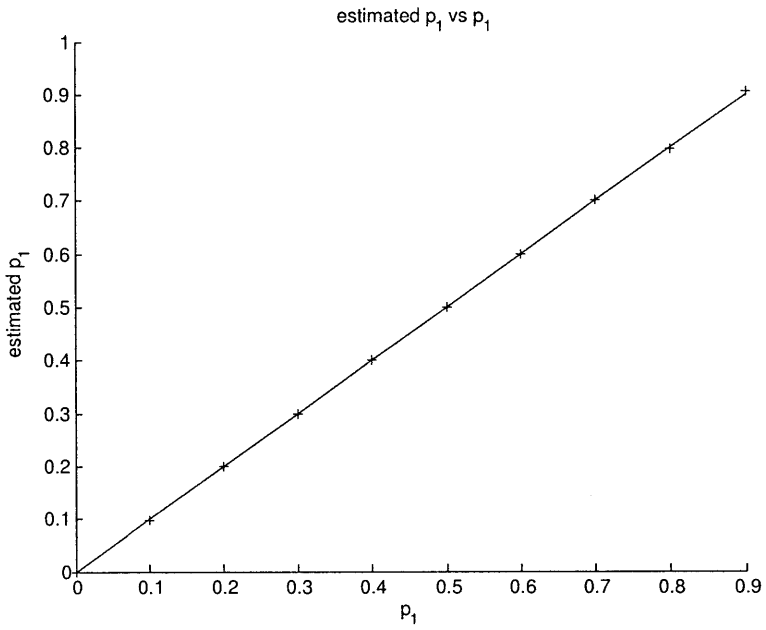


Figure 2.7: Estimated p_1 vs p_1 using the centroid of the region of PQ-space corresponding to observed $\bar{i}[n]$.

2.3 Region of PQ-space corresponding to quantized samples

Given a particular set of quantized samples $\bar{i}[n]$, the goal is to determine the corresponding region in PQ-space that contains all possible values of the p_k's and q_k's that could produce the observed quantized samples. Call this region H. As shown in Lemma 1, H is a convex polytope, and so is completely determined by its vertices. Thus, a potential goal could be to determine the coordinates of these vertices. This is rather undesirable due to the fact that H is an N-polytope and may have a large number of vertices. Fortunately, it is unnecessary to determine the vertices of H because, as noted above, we only require the intersection of H with the (comparatively) low dimensional subspace given by the p_k's and q_k's that are not identically zero. Let Z denote this subspace. Let W denote the number of p_k's and q_k's that are not identically zero (Z is

a W dimensional subspace). Let R denote this intersection. Clearly R is also a convex polytope (because it is the intersection of a convex polytope with a subspace) but it is of a potentially lower dimension (it is only a W-polytope). Thus, the goal will be to find the vertices of R.

The following analysis will consider only non-empty regions R. Empty regions R can be safely ignored because they arise from polytopes H that do not intersect Z. These polytopes are of no interest because, by assumption, we encounter only quantized samples of waveforms described by points in Z.

As a first step, notice that given $\overline{i}[n]$, it is easy to find a single point in R. This can be done by first producing any set of unquantized samples, $\hat{i}[n]$, that quantize to $\overline{i}[n]$ (that is to say, $\overline{i}[n] = Q(\hat{i}[n]), \forall n$) and satisfy the constraint that the point must lie in Z. This is easy because it only requires finding a feasible solution to a system of linear inequalities in W (not N) variables. Next, take the DFT of $\hat{i}[n]$ to produce the $\hat{p}_k$'s and $\hat{q}_k$'s. These $\hat{p}_k$'s and $\hat{q}_k$'s specify a point that lies in H because, by construction, the $\hat{p}_k$'s and $\hat{q}_k$'s correspond to $\hat{i}[n]$, which quantizes to $\overline{i}[n]$. This point is in Z and so the point lies in $Z \cup H = R$.

This point is not necessarily the centroid of R (and is, of course, quite unlikely to be the centroid of R), but it is still a point in R. Thus, this point could serve as an estimate of the true spectral content if one wished to avoid the extra computational cost of determining the vertices and eventually the centroid of R. While this estimate is generally not as good as using the centroid, it is, in general, still a significant improvement over using the $\overline{p}_k$'s and $\overline{q}_k$'s as estimates for the p_k's and q_k's.

Next, an algorithm will be presented that determines the vertices of R from any point in R. The rough idea of the algorithm is to first find a single vertex of R, then

find all of its neighboring vertices, then all of their neighboring vertices and so on until every vertex is found. This algorithm will require the use of two subroutines: FIND-FIRST-VERTEX(y) and FIND-NEIGHBORS(y). These subroutines will be described first.

The FIND-FIRST-VERTEX(y) subroutine finds a single (arbitrary) vertex of R, where $y \in R$. As noted above, R is a (bounded) convex polytope. By definition, this means that R is the intersection of a collection of half-spaces. Each half-space corresponds to the region of space that satisfies the set of linear constraints that each of the N points $i[n]$ lie in the quantization interval $\bar{i}[n]$. To be precise, define values $l_1, \ldots, l_N$ and $u_1, \ldots, u_N$ such that the lower and upper boundary points of the quantization interval whose representation point is $\bar{i}[n]$ are l_n and u_n, respectively. Then we have the following $2N$ linear constraints:

$$i[n] > l_n, \forall n \tag{2.7}$$

and

$$i[n] \leq u_n, \forall n. \tag{2.8}$$

Due to the fact that we have the constraint that each $i[n]$ must be less than each upper boundary point but strictly greater than each lower boundary point, R is neither closed nor open, because it contains only part of its boundary (the portion that corresponds to spectral content for which the corresponding $i[n]$ satisfies $i[j] = u_j$, for some j). However, because we are only interested in the vertices of the polytope, we

can consider only the closure of R (the smallest closed set that contains R). This set is defined by the following $2N$ linear constraints, which are identical to the above $2N$ linear constraints with the exception that all inequalities have been made non-strict. In the remainder of this analysis, R will refer to the closure of the original R defined above.

$$i[n] \geq l_n, \forall n \tag{2.9}$$

and

$$i[n] \leq u_n, \forall n. \tag{2.10}$$

Interior points of R are points at which all inequalities are strictly satisfied (no inequality is satisfied with equality). The boundary points of R are those points for which at least one of the constraints are satisfied with equality. Vertices of R are local maxima for the number of constraints satisfied with equality (infinitessimal movement from a vertex in any direction that remains in R will decrease the number of satisfied constraints). FIND-FIRST-VERTEX locates a vertex by starting with any point in R and moving that point in such a way as to satisfy increasingly many constraints with equality. Let the constraints be numbered $1, \ldots, 2N$ in arbitrary order, let Z_i denote the subspace of R in which constraint i is satisfied with equality. This algorithm is shown in pseudocode below.

FIND-FIRST-VERTEX(y)

$x \leftarrow y$

$S \leftarrow R$

for $i = 1$ to $2N$ **do**

  if $x \in Z_i$ **then**

    $S \leftarrow S \cap Z_i$

  end if

end for

repeat

  select arbitrary $s \in S$

  $x' \leftarrow x + ks$ where $k > 0$, k is the smallest value such that x' lies in the boundary of R $\{x'$ is the translation of x that satisfies at least one new constraint$\}$

  for $i = 1$ to $2N$ **do**

    if $x' \in Z_i$ and $x \notin Z_i$ **then**

      $S \leftarrow S \cap Z_i$

    end if

  end for

  $x \leftarrow x'$

until $dim(S) = 0$

return x

The algorithm keeps track of a single point x, and a space S, which are updated by a series of moves. x is initialized to the point y known to be in R and S is initialized to R. Next, S is updated to be the subspace of R that is the intersection of all Z_i for which x satisfies constraint i with equality, by intersecting S with Z_i. The space S represents the space of directions in which x can be translated such that every constraint initially satisfied with equality is still satisfied with equality after the translation. We then move

along some direction $s \in S$ until at least one new constraint is satisfied with equality. x and S are then updated. We continue moving x until $dim(S) = 0$, which corresponds to a point at which any infinitessimal translation of x in any direction (in R) would cause some constraint that is currently satisfied with equality to no longer be satisfied with equality. Thus, when $dim(S) = 0$, we are at a local maxima for the number of satisfied constraints, and so x is a vertex of R. Equivalently, the condition $dim(S) = 0$ can be viewed as expressing the fact that the space that satisfies the Z_i constraints found in each iteration is a 0-dimensional space (a point).

The purpose of $FIND - NEIGHBORS(y)$ is to find all vertices that are neighbors of the vertex y. By definition, two vertices x and y are neighbors if they are connected by an edge. Thus, we can find all neighbors of y by moving along each edge incident to y until a new vertex is reached. Moving along an edge is accomplished by selecting a constraint that is satisfied with equality at y and relaxing it (allowing it to be satisfied with inequality) by moving along the edge. The algorithm is shown in pseudocode below.

FIND-NEIGHBORS(y)

$S \leftarrow \{\}$

$L \leftarrow \{\}$

for $i = 1$ to $2N$ **do**

  if $y \in Z_i$ **then**

    $S \leftarrow S \cup Z_i$

  end if

end for

for all Z_i in S **do**

  if $dim(S \backslash Z_i) == 1$ **then**

Assign s to be an element in $S \backslash Z_i$ such that moving y in the direction s keeps constraint i satisfied.

$x \leftarrow y + ks$ where $k > 0$, k is the smallest value such that x lies in the boundary of R $\{x$ is the translation of y that satisfies at least one new constraint$\}$

$L \leftarrow L \cup \{x\}$

end if

end for

return L

This algorithm makes use of two sets: S consists of the constraints satisfied with equality at y, L is the set of neighbors of y that is being determined. The algorithm first builds S by checking which constraints are satisfied with equality. Then, it finds each edge incident on L by using the fact that each edge is a one dimensional subspace given by $S \backslash Z_i$, for some Z_i. It then finds a neighboring vertex x by moving along that edge and then adds x to L.

Finally, we can describe the algorithm $FIND - ALL - VERTICES(y)$ which finds all vertices of R given some $y \in R$. This algorithm is shown in pseudocode below.

FIND-ALL-VERTICES(y)

$V \leftarrow \{\}$

$B \leftarrow \{FIND - FIRST - VERTEX(y)\}$

repeat

$V \leftarrow V \cup B$

$L \leftarrow \{\}$

for all $b \in B$ **do**

$L \leftarrow L \cup (FIND - NEIGHBORS(b) \cap V^{\perp})$

end for

$B \leftarrow L$

until $|B| = 0$

This algorithm builds up a set V of vertices of R. It operates in a series of stages, where in each stage it operates on newly discovered vertices, stored in B. Initially, B contains the first vertex of V, found by FIND-FIRST-VERTEX. In each stage, the elements of B are added to V. Then, the set L is constructed which consists of all neighbors of vertices in B that are not already in V. Then, B is set to L and the cycle repeats until no new vertices are found. This process terminates because there are a finite number of vertices and each vertex can only be newly discovered once. It finds all vertices because, after iteration i, all vertices that are at distance $\leq i$ have been found, and every vertex is a finite number of steps from every other vertex (the graph with the vertices and edges of R is connected).

A Matlab implementation of the above algorithm is included in Appendix A.

2.4 Calculations from regions of PQ-space

The previous section illustrated how to find the vertices of a polytope $R = H \cap Z$, where H is an N-polytope in PQ-space that contains all points with the same quantized samples $\bar{i}[n]$ and Z is a W dimensional subspace. Using the vertices of R, we can compute both the volume of R, the centroid of R, and the maximum distance between the centroid

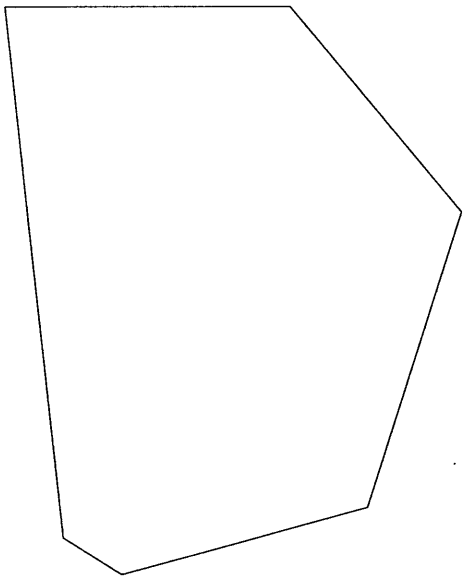


Figure 2.8: Example polygon.

and any point in R. The centroid is useful because it can be used as a relatively accurate prediction of the true spectral content of the unknown $i[n]$ that quantizes to the known $\bar{i}[n]$. The volume and maximum distance are useful for analyzing the accuracy of the prediction algorithm.

In order to compute the volume of the polytope R, we can partition R into a set of simpler polytopes, and take the sum of their volumes. Before solving this problem in arbitrary dimension, consider the following example in 2 dimensions. Fig. 2.8 shows a convex 2-polytope (polygon) R.

R can be partitioned into a set of triangles by selecting an arbitrary point x in the interior of R and drawing line segments from x to each vertex of R, as shown in Fig. 2.9. The area of R can then be computed by computing the area of these triangles and summing the results.

This concept can be generalized to an arbitrary W-polytope R by partitioning R into W-simplexes. A W-simplex is an W-dimensional convex polytope with $W + 1$

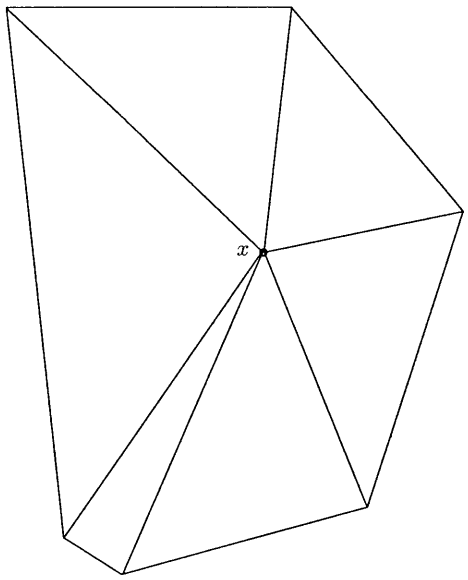


Figure 2.9: Example polygon with interior point.

vertices. It can be thought of as the generalization of a triangle to higher dimensions. This partitioning is accomplished by again selecting an arbitrary point x in the interior of R and drawing line segments to each vertex of R. These line segments are the edges of the W-simplexes. The volume V of a W-simplex with vertices $\{v_0, \ldots, v_W\}$ is given by

$$V = \frac{1}{W!} det \begin{pmatrix} v_1 - v_0 & v_2 - v_0 & \ldots & v_{W-1} - v_0 & v_W - v_0 \end{pmatrix}, \tag{2.11}$$

where the determinant is taken on an $W \times W$ whose jth column consists of the elements of $v_j - v_0$.

The centroid of R can be also be computed through a decomposition into simplexes. To be precise, we can partition R into a set of simplexes, as above, compute the centroids of those simplexes, and then take the weighted average of those centroids (weighted by the volume of the centroid). This weighted average is the centroid of R.

The centroid C of a W-simplex with vertices $\{v_0, \ldots, v_W\}$ is given by

$$C = \frac{1}{W+1} \sum_{j=0}^{W} v_j. \tag{2.12}$$

The maximum distance D between any point in the region and the centroid C, provides a bound on the absolute maximum error between the actual spectral content of a point that produced quantized samples $\bar{i}[n]$ and the estimated spectral content C. To determine D, we can use the fact that the point in R at maximum distance from C will be one of the vertices of R (because R is a bounded polytope). Thus, D is given by

$$D = \max_{j \in [0,W]} |C - v_j|. \tag{2.13}$$

The computations discussed above are only a sample of the sort of computations possible about properties of the region R using only the vertices of that region. The determination of the vertices of R, using the algorithm of the previous section, conveniently allows the efficient computation of a variety of other useful quantities, such as the maximum distance between any point in R and the centroid and the expected distance between a randomly chosen point (according to some known distribution) and the centroid.

Chapter 3

Cross Estimation

3.1 Introduction

The previous chapter considered the question of accurately estimating the spectral content (the collection of p_k's and q_k's) of a signal $i[n]$ given the spectral content $\overline{p}_k$ and $\overline{q}_k$ of the quantized signal $\overline{i}[n]$. This was accomplished by using additional information about the structure of $i[n]$, specifically, the fact that the true $i[n]$ consists of non-zero spectral content at a (known) limited set of frequencies. This chapter will consider the related question of estimating one subset of p_k's and q_k's from knowledge of another subset. In general, of course, nothing at all can be said about the value of any particular p_j or q_j given knowledge of any other p_k's and q_k's because all of the p_k's and q_k's are independent. However, much as was the case in the previous chapter, there will often be additional information about the structure of $i[n]$ that will allow this cross-estimation.

Before discussing how to actually perform this sort of estimation, it is useful to consider a particular problem that motivates the desire to be able to use the values of

known spectral envelopes to estimate unknown spectral envelopes. In many settings, the observed current signal $i[n]$ will be an aggregate current signal. That is to say,

$$i[n] = \sum_j i_j[n],$$

where each $i_j[n]$ is the current drawn by a single electrical load. This situation arises when monitoring a (potentially large) collection of electrical loads by taking measurements of only the aggregate current. In this setting, one often desires to know the spectral content of an individual $i_j[n]$. If we let $p_{j,k}$ and $q_{j,k}$ denote the kth spectral envelope of $i_j[n]$, then by the linearity of the DFT, we have

$$p_k = \sum_j p_{j,k} \text{ and } q_k = \sum_j q_{j,k}.$$

Thus, the goal here would be to use knowledge of the p_k's and q_k's to estimate the individual $p_{j,k}$'s and $q_{j,k}$'s. Fortunately, the current drawn by different types of electrical loads will often consist of different sets of spectral content [1]. For example, consider the case when $i[n]$ consists of the sum of two different individual current signals, $i_1[n]$ and $i_2[n]$, where the only non-zero spectral content of $i_1[n]$ is $p_{1,1}$ and $q_{1,1}$ and the only non-zero spectral content of $i_2[n]$ is $p_{2,1}$ and $p_{2,5}$. Here, the sets of non-zero spectral content are only partially overlapping. Thus, $q_1 = q_{1,1}$ and $p_5 = p_{2,5}$, and so knowledge of the aggregate q_1 and p_5 allows the corresponding spectral content of the individual loads to be determined. Unfortunately, $p_1 = p_{1,1} + p_{2,1}$, so it is not immediately clear how to use the aggregate value p_1 to determine the individual values $p_{1,1}$ and $p_{2,1}$. This chapter will attempt to answer this question by using the attainable $q_{1,1}$ to estimate $p_{1,1}$ and $p_{2,5}$ to estimate $q_{2,5}$, using additional information about the structure of $i_1[n]$ and $i_2[n]$.

3.2 Usable Constraints

There are many different sorts of constraints that one could apply to a signal $i[n]$. This chapter will examine a method that uses constraints of the form

$$\sum_k a_k s_k = 0, \tag{3.1}$$

where

$$s_k = p_k + iq_k.$$

Here, for convenience, we express spectral content as complex values s_k rather than separately as real and imaginary components p_k and q_k. Each $a_k \in \mathbb{Z}[\zeta_N]$, where ζ_N is a primitive Nth root of unity (that is to say, $\zeta_N^N = 1$ but $\zeta_N^j \neq 1$ for $j < N$), and $\mathbb{Z}[\zeta_N]$ denotes adjoining ζ_N to the integers $\mathbb{Z}$. Thus each $a \in \mathbb{Z}[\zeta_N]$ is of the form

$$a = \sum_{j=0}^{N-1} b_j \zeta_N^j,$$

where $b_j \in \mathbb{Z}$. We restrict constraints to this form to allow an efficient and accurate solution method, discussed in section 4 of this chapter, that exploits the properties of cyclotomic fields.

While this family of constraints certainly doesn't capture every possible constraint that could exist on a signal $i[n]$, it is still a rather general class that includes many useful constraints. For example, the constraint $s_k = 0$, for any particular k, is clearly in this class (this corresponds to setting $a_k = 1$ and $a_j = 0$, $\forall j \neq k$). Similarly, the constraint $i[j] = 0$, for any particular point sample j is in the class because the Fourier synthesis equation expresses $i[j]$ as a linear combination of the s_k, where the coefficients are powers

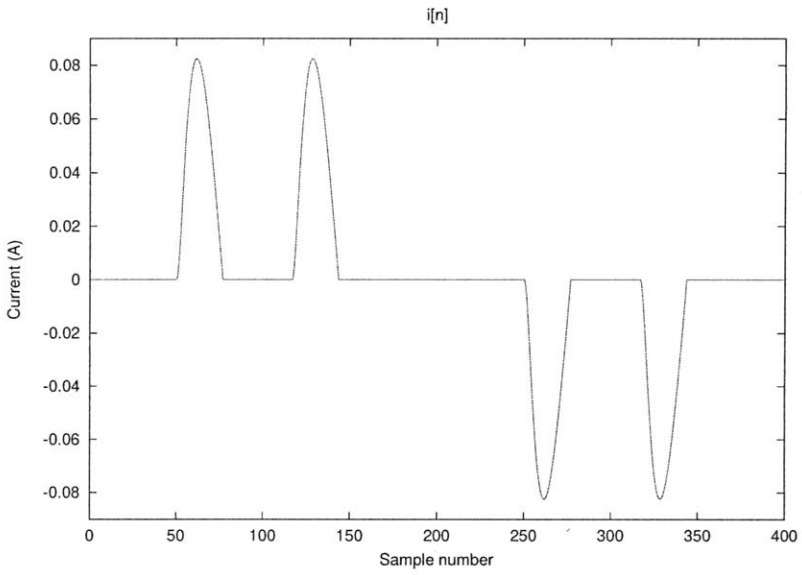


Figure 3.1: The current drawn by a Variable Speed Drive (VSD) over one line-cycle.

of some ζ_N. Similarly, any constraint of the form $\sum_{j \in J} c_j i[j] = 0$, for any subset of sample indices J and any $c_j \in \mathbb{Z}[\zeta_N]$ are also in this class. This last family of constraints includes "symmetry" constraints, such as the statement $i[j] = i[l]$ or $i[j] = -i[l]$, for any sample indices j, l.

It should be noted that we could have instead reasonably defined each a_k to be an element of $\mathbb{Q}[\zeta_N]$ rather than $\mathbb{Z}[\zeta_N]$. However, restricting ourselves to $\mathbb{Z}[\zeta_N]$ is without loss of generality because any constraint where $b_j \in \mathbb{Q}$ could trivially be transformed into a constraint with only integer coefficients by multiplying by a common denominator.

This class of constraints is selected because it permits an efficient and accurate solution method, while still being general enough to capture real world constraints. To demonstrate the generality of these constraints, consider the following example waveform, shown in Fig. 3.1, that shows the current drawn by a variable speed drive (VSD).

This waveform clearly allows many constraints of the above form to be applied. For example, the large regions of zeros allow constraints of the form $i[j] = 0$, and the symmetry of the non-zero regions allow symmetry constraints. Moreover, it is known [6] that, for this waveform, even harmonics and the so-called "triplen" harmonics (multiples of 3) are approximately zero, which allows constraints of the form $s_k = 0$, for k a multiple of 2 or 3. For clarity, this particular concrete example will be used throughout the chapter to illustrate the various techniques considered.

3.3 A First Attempt at a Solution

The goal will be to express a single unknown spectral envelope s_r in terms of a collection of known spectral envelopes s_j, for $j \in J$. As usual, we consider samples $i[n]$ of some periodic waveform $i(t)$. Unlike in the previous chapter, in this situation we do not actually take as input $i[n]$ but rather just the s_j, $\forall j \in J$. We also assume that we have sufficient knowledge of $i(t)$ to generate constraints. Rather than viewing the number of samples per period, N, as some fixed value determined by sampling, here we can set N based on the number of constraints we desire. The idea is that we know the general form of $i(t)$, and so can correctly write down a family of constraints for any sampled signal $i[n]$ consisting of N samples, for any N. For example, if faced with the current waveform $i(t)$ of the VSD, Fig. 3.1, we can immediately determine which indices j should have the constraint $i[j] = 0$, for any number of samples N by simply checking if the jth of N samples would land in a region of zeros. The key point is that we can set N arbitrarily.

Every constraint of the form expressed in (3.1) sets a linear combination of the s_k's to 0, where each coefficient $a_k \in \mathbb{Z}[\zeta_N]$. Thus, we can form a matrix equation of the

form

$$0 = AS, \tag{3.2}$$

where S is a vector of the s_k's, and A is a matrix with entries in $\mathbb{Z}[\zeta_N]$ where each row represents a single constraint. We can order the entries of S in any order; place the unknown s_r first, and the known s_j, $j \in J$ last, with all other spectral envelopes in arbitrary order. With this ordering of S, the first column of A corresponds to coefficients multiplying s_r and the last $|J|$ columns of A correspond to coefficient multiplying s_j, $j \in J$. We wish to set A to be of size $M \times (M + |J|)$, for some M. This is desirable because if we place A in reduced row echelon form (RREF), the first row of the resulting matrix will express an equation of the form

$$s_r + \sum_{j \in J} c_j s_j = 0, \tag{3.3}$$

which gives an expression for the unknown s_r in terms of the known s_j, $j \in J$, as desired, where $c_j \in \mathbb{Q}[\zeta_N]$.

To assure that we can form the $M \times (M + |J|)$ matrix A, we will further make the assumption that $i(t)$ is bandlimited, that is to say, $i(t)$ contains no spectral content outside of some finite band. This means that, for any N, there exists a single constant N_0 such that at most N_0 of the s_k of $i[n]$ will be non-zero. We will also make the assumption that $i(t)$ has a region of zeros, some sort of symmetry, or any other structure that allows a number of constraints of the above form, that increases with N, to be written. The number of such valid constraints grows with N because, if for example, $i(t)$ has a region of zeros, then as N increases, more and more sample points will fall in that region. Consequently, as N increases, the number of constraints on $i[n]$ increases but the number of non-zero s_l does not increase past some finite limit.

To be precise, let N_0 denote the finite limit on the number of non-zero spectral envelopes implied by the fact that $i(t)$ is bandlimited. Then consider increasing N, starting at N_0. As N increases, increasingly many s_k become defined, but all the "new" $s_k = 0$. At the same time, we have increasingly many constraints on $i[n]$. This means that the total number of constraints on the s_k increases with N faster than the number of defined s_k (every new s_k introduced comes with the constraint $s_k = 0$, but we also add other new constraints, such as zero constraints on $i[n]$, or symmetry constraints). Thus, at some point, we have $N - |J|$ constraints on the N spectral envelopes. We can then write the matrix equation

$$0 = AS$$

where A is a $M \times (M + |J|)$ matrix, with $M = N - |J|$.

Many of the constraints are simply of the form $s_k = 0$, and so we can delete each column corresponding to such an s_k, remove the entry from S that corresponds to s_k and delete the row of A that corresponds to the constraint. This will improve the speed of subsequent computations on the matrix by decreasing its size, without hurting the resulting accuracy. It should be noted that this idea does not only work on a single N, but rather all sufficiently large N because as N increases further, we only get more constraints relative to the number of spectral envelopes. Whenever we have "extra" constraints (that is to say, more constraints than would fit in a $M \times (M + |J|)$ matrix), we can simply not use the extra constraints. In particular, we can choose to ignore constraints of the form $s_k = 0$ when we have extra constraints. This is desirable as the constraints $s_k = 0$ are sometimes only approximate because the waveform $i(t)$ is only approximately bandlimited. One could expect accuracy of the resulting estimation to increase with N because as N gets larger and larger, we are able to use (relatively) fewer constraints of the form $s_k = 0$

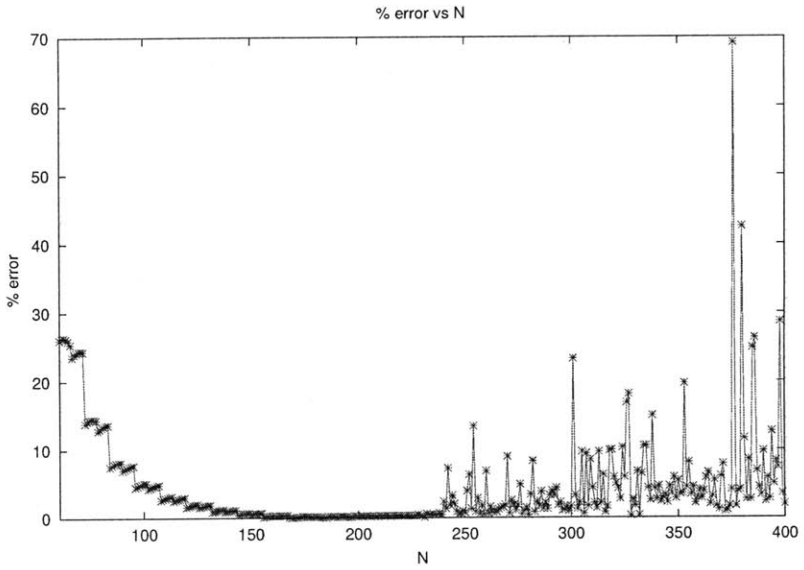


Figure 3.2: Percent error using s_5 and s_7 to estimate s_1.

This idea was applied to samples $i[n]$ of the current waveform $i(t)$ depicted in Fig. 3.1 to estimate s_1 from knowledge of s_5 and s_7. In Fig. 3.2, the error of the resulting estimation is plotted as a function of N. While this does obtain somewhat low error, and the error decreases with N initially, as expected, the error actually increases without bound for very large N. This is due to a numerical problem. The computation to put A in RREF, by performing Gaussian elimination involves floating point division by numbers that get smaller with increasing N. This substantially limits the effectiveness of this technique. To avoid this problem, the following section will consider a different solution method in which all computation is done, effectively, over the integers. This will completely eliminate numerical problems.

3.4 A Refined Solution Using Cyclotomic Fields

As noted in the previous section, transforming the constraint matrix into RREF by computing with floating point arithmetic is numerically unstable. Fortunately, this problem can be completely avoided by recognizing that the elements in the matrix, initially as well as at every step of the RREF computation, have a certain special property: they are elements of a cyclotomic field. A cyclotomic field is simply an algebraic number field generated over $\mathbb{Q}$ (the rationals) by a primitive root of unity. Algebraic number fields (called number fields by some authors) are finite (and therefore algebraic) extensions of $\mathbb{Q}$. Elementary properties of the cyclotomic fields can be found, for example, in [3]. The N^{th} cyclotomic field is simply $\mathbb{Q}[\zeta_N]$ (this denotes adjoining ζ_N to $\mathbb{Q}$). Any element $y \in \mathbb{Q}[\zeta_N]$ can be expressed in the form

$$y = \sum_{j=0}^{N-1} c_j \zeta_N^j, \tag{3.4}$$

with $c_j \in \mathbb{Q}$. Clearly, every element of the matrix A above is an element of $\mathbb{Q}[\zeta_N]$. Moreover, since the cyclotomic field is a field (with the usual arithmetic operations) it is closed under addition, subtraction, multiplication and division. As the process of performing Gaussian elimination only involves these arithmetic operations, we see that the matrix, at any point during the computation, will only consist of elements from a cyclotomic field. Clearly, elements of the cyclotomic field can be represented exactly (by, for example, storing the rational coefficients c_j that define each element y), and so it is possible to perform the computation exactly.

A straightforward way to perform computations over the cyclotomic field would be to store the collection of rational coefficients c_j that represent a given element in each entry of the matrix. Then, perform Gaussian elimination as usual, substituting the

cyclotomic field operations in place of arithmetic on scalar quantities. A slightly different approach will be taken here for computational efficiency reasons.

There is a natural isomorphism between the N^{th} cyclotomic field and $\mathbb{Q}[X]/f_N(X)$, where $\mathbb{Q}[X]$ denotes the ring of all polynomials in one variable with rational coefficients, and $f_N(X)$ denotes the N^{th} cyclotomic polynomial [3]. The N^{th} cyclotomic polynomial is defined by

$$f_N(X) = \prod_{\omega \in \Omega} (X - \omega),$$

where Ω consists of all primitive N^{th} roots of unity in $\mathbb{C}$. Thus, we can view each element of A as an equivalence class of polynomials over a single formal parameter. That is to say, each particular element of A can be viewed as the set of all polynomials with rational coefficients that are equivalent, modulo $f_N(X)$, to a single specific polynomial (this specific polynomial is different for different entries of the matrix). To better understand this isomorphism, notice that any $y \in \mathbb{Q}[\zeta_N]$ is given by (3.4) as a linear combination of powers of ζ_N. In some sense, we can view y as being the value of a polynomial $g(X) \in \mathbb{Q}[X]$ evaluated at ζ_N, where

$$g(X) = \sum_{j=0}^{N-1} c_j X^j.$$

The coefficients of the polynomial are the same as the coefficients used to define y. Moreover, we could view y as being the value of any polynomial $h(X) \in \mathbb{Q}[X]$ at ζ_N where $h(X) = g(X) + f_N(X)k(X)$, with $k(X) \in \mathbb{Q}[X]$ being arbitrary because $f_N(X)$, the N^{th} cyclotomic polynomial, has ζ_N as a root. The family of $h(X)$ is simply the equivalence class of polynomials (in $\mathbb{Q}[X]$) that are congruent to $g(X)$ modulo $f_N(X)$. While this helps make clear the structure of the isomorphism, it is important to remember that the X in each polynomial is only a formal parameter; it will not take any values.

Using this idea, we can store in each entry of A an arbitrary lift of the equivalence

class of polynomials represented by that entry (that is to say, store any single polynomial in the equivalence class). We can store a polynomial by storing its coefficients. Notice that $f_N(X)$ is of degree $\phi(N)$ [3], where $\phi(N)$ is Euler's totient function and is defined to be the the number of integers $j < N$ where j is relatively prime to N. Thus, we have a slight reduction in storage over the initial scheme of storing the coefficients expressed in (3.4). However, since $\phi(N) = \Theta(N)$, this is actually not an asymptotic improvement in storage, but still might be useful in practice. To perform Gaussian elimination, we simply replace the ordinary arithmetic operations that Gaussian elimination would perform on scalars with the corresponding operations on polynomials (addition becomes addition of coefficients, multiplication becomes convolution of coefficients, and so on) with the added fact that we perform operations modulo $f_N(X)$. Again, we see a slight computational improvement by using the polynomial representation rather than the initial representation because we are only operating on $\phi(N)$ coefficients rather than N coefficients. This is again not an asymptotic improvement, but might still be of value in practice. In any case, addition and subtraction are $O(N)$, and multiplication and division are $O(N^2)$ by the naive algorithms. Since Gaussian elimination involves $O(N^3)$ arithmetic operations, we have a runtime bound of $O(N^5)$, which is still reasonable due to the relatively small N involved. Multiplication and division will be improved to $O(N \log N)$ in the next section by using the Number Theoretic Transform and properties of multiplying and dividing polynomials. This will improve the runtime bound to $O(N^4 \log N)$.

The above algorithm performs all computations exactly to produce a relation of the form $s_r = \sum_{j \in J} b_j s_j$, where $b_j \in \mathbb{Q}\zeta_N$, which expresses unknown s_r in terms of known s_j. Of course, actually evaluating s_r from the s_j will involve computing this sum with floating point arithmetic (because s_j will likely not be elements of $\mathbb{Q}\zeta_N$ but rather arbitrary values in $\mathbb{C}$). Fortunately, this only involves a small number of floating point

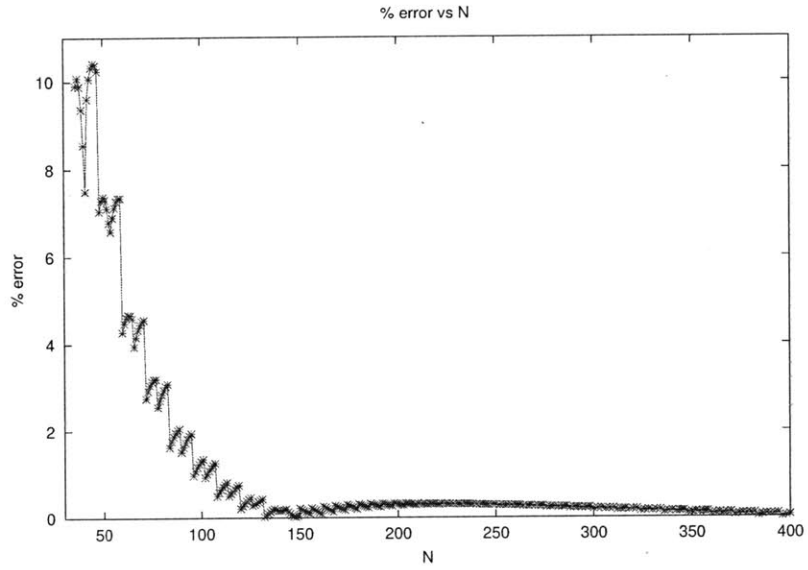


Figure 3.3: Percent error using s_5 and s_7 to estimate s_1 with the refined method.

additions and multiplications and so does not exhibit the numerical instability of the initial solution technique.

This procedure was applied to samples $i[n]$ of the current waveform $i(t)$ depicted in Fig. 3.1 to estimate s_1 given s_5 and s_7, as in the previous section. The results are shown in Fig. 3.3. As can be seen, there is no longer a numerical instability. The implementation was done in GP/PARI; the code is included in Appendix B.

3.5 Speed Improvement Using the Number Theoretic Transform

While the algorithm presented in the previous section is sufficiently fast to be practical, there is still room for improvement. This section will consider a method to

improve the speed of the basic arithmetic operations of multiplication and division. Multiplication and division of polynomials involves convolution and deconvolution, respectively, of coefficients. This immediately suggests using a procedure like the Fast-Fourier Transform (FFT). The convolution theorem [2], states that, for $x = x_1, x_2, \ldots, x_N$ and $y = y_1, y_2, \ldots, y_N$, we have

$$FFT(x * y) = FFT(x)FFT(y).$$

As the FFT can be computed in $O(N \log N)$ time, this immediately yields a $O(N \log N)$ algorithm for multiplication and division of polynomials. Specifically, given polynomials $f(X) = \sum_j f_j X^j$ and $g(X) = \sum_j g_j X^j$, we have $f(X)g(X) = h(X) \equiv \sum_j h_j X^j$, where $h_j = \sum_k f_k * g_{j-k}$, which is a convolution. Thus, we can multiply $f(X)$ and $g(X)$ by taking the FFT of the coefficients of $f(X)$ and $g(X)$ separately, multiplying the FFTs elementwise, and computing the inverse transform (again, using an FFT). The result will be the coefficients of $h(X)$. Division functions in an analogous fashion, with the only modification being that we divide FFTs elementwise.

An immediate problem with the above scheme is the fact that it involves computing FFTs with floating point operations. Since our goal is perform all computations exactly, we would have to determine the proper exact representation of coefficients returned by the above procedure. While this is, in principle, possible, it adds unnecessary extra work. As an alternative, consider the Number Theoretic Transform (NTT).

Given a sequence $x = x_1, \ldots, x_N$ where $x_j \in \mathbb{Z}$, the NTT of x is a sequence $X = X_1, \ldots, X_N$ where

$$X_k = \sum_n x_n \omega^{nk} \mod p, \tag{3.5}$$

and

$$x_n = \sum_k X_k \omega^{-nk} \mod p, \tag{3.6}$$

where $s \in \mathbb{N}$ is a free parameter that will be set later, $p = sN + 1$ is a prime, and $\omega = \eta^s$, where η is a primitive sN^{th} root of unity, modulo p. The NTT also obeys the convolution theorem, but has the advantage that all computation is done over the integers. One can also immediately define a "Fast" NTT, analogous to the FFT, which uses an identical divide and conquer approach to compute an NTT in $O(N \log N)$. Thus, we can use an NTT in place of the FFT in the above procedure to enable the fast computation of multiplication and division.

To do this correctly, we must set the prime p to be larger (by a factor of 2) than any coefficient in the polynomials input to multiplication and division, as well as any coefficient in the output polynomial (the value of each coefficient in the output polynomial can trivially be bounded in terms of the coefficients of the input polynomial). This is done by setting s appropriately. This is necessary to assure that if we take the representative in $[-\frac{p-1}{2}, \frac{p-1}{2}]$ of each congruence class, we will obtain the correct coefficient (this is just saying there there is no ambiguity introduced by working modulo p; that is to say, p is large enough so that knowing the value of the coefficient modulo p immediately yields the value of the coefficient). Code to perform these calculations is included in Appendix B.

3.6 Ring of Integers of a Cyclotomic Field

This section will consider an alternate scheme for estimating an unknown s_r in terms of known s_j, $j \in J$. We begin with some terminology from algebraic number theory; see, for example [4]. Let K denote a number field (in our application, K will be

a cyclotomic field, but the following definitions apply to all number fields). We say an element $x \in K$ is *integral* over a ring B if we have an equation of integral dependence:

$$x^n + b_{n-1}x^{n-1} + \ldots + b_1 x + b_0 = 0, \tag{3.7}$$

where $b_i \in B$, $\forall i$. This is simply the statement that x is a root of a monic polynomial with coefficients in B. We call the collection of elements in K that are integral over the ring $\mathbb{Z}$ the *integers* of K. These elements form a ring (with the usual addition and multiplication operations), but not a field (in general, we cannot divide elements). This ring is called the *ring of intgers* of K.

For a cyclotomic field, the ring of integers is simply $\mathbb{Z}[\zeta_N]$ [3], and so every element of the constraint matrix A (see (3.2)) is an element of the ring of integers of a cyclotomic field. The previous algorithm uses Gaussian elimination to transform A into RREF, at which point the first row of the matrix corresponds to the equation $s_r + \sum_{j \in J} c_j s_j = 0$, where $c_j \in \mathbb{Q}[\zeta_N]$. Here, the idea will be to work in the ring $\mathbb{Z}[\zeta_N]$ and ultimately produce an equation $d' s_r + \sum_{j \in J} d_j s_j = 0$, where $d_j \in \mathbb{Z}[\zeta_N]$, $\forall j \in J$ and $d' \in \mathbb{Z}[\zeta_N]$.

Of course, we cannot simply use Gaussian elimination because that requires division, which cannot (in general) be done in a ring. The idea will be to use a similar process where we skip the step of dividing a row by its leading element (this is the only step of Gaussian elimination that involves division). To be precise, in ordinary Gaussian elimination, we operate column by column, transforming the matrix so that each column has only a single 1 and all other entries 0. For each column, we select a row with a non-zero element in that column to operate on, call this row m. The sequence of steps performed by standard Gaussian elimination, for row m, is shown in pseudocode below. In the following, let $leading(m)$ return the index of the first non-zero entry in row m, let

R and C denote the number of rows and columns, respectively, of the matrix, and let a_{ij} denote the entry in row i column j.

$c \leftarrow leading(m)$

$p \leftarrow a_{mc}$

for $j = 1$ to C **do**

$a_{mj} \leftarrow a_{mj}/p$

end for

for all $i \in [1, R] \setminus \{m\}$ **do**

$q \leftarrow a_{ic}$

for $j = 1$ to C **do**

$a_{ij} = a_{ij} - a_{mj}q$

end for

end for

This will be modified to the following:

$c \leftarrow leading(m)$

$p \leftarrow a_{mc}$

for all $i \in [1, R] \setminus \{m\}$ **do**

$q \leftarrow a_{ic}$

for $j = 1$ to C **do**

$a_{ij} = a_{ij}p - a_{mj}q$

end for

end for

In the original Gaussian elimination algorithm, we operate on row m by first dividing each entry of row m by the leading element, then, for every row $i \neq m$, we subtract a multiple q of row m from row i, where q is simply a_{ic}, the element in row i in the same column as the leading element of row m. This has the effect of clearing column c, except for a_{mc}, which is set to 1. Every step of this process preserves the validity of the system of equations because we are only either dividing a row by a constant or subtracting a multiple of one row from another row. This indeed preserves the validity of the system of equations represented by the matrix because each row i of the matrix corresponds to an equation $\sum_j a_{ij}s_j = 0$, and thus dividing by a constant only divides all coefficients by the same constant; similarly, subtracting a multiple of one row to another corresponds to subtracting a multiple of one equation from another. The only changes are that we now do not divide row m by its leading element but instead multiply each row i by the leading element of row m and then subtract the same multiple q of row m from row i. We are allowed to multiply row i by the leading element of row m (or, in fact, by any constant) because again row i represents an equation of the form $\sum_j a_{ij}s_j = 0$, which is unaffected by multiplying the coefficients by any non-zero constant.

All operations in this new scheme can be done over a ring, and so the above algorithm could indeed be used to produce a relation $d's_r + \sum_{j\in J} d_j s_j = 0$, as desired. One significant problem, however, is that if the above algorithm is used as described, the coefficients of the polynomials stored in each entry of the matrix will become extremely large. In such a situation, it will no longer be appropriate to treat the individual arithmetic operations on coefficients as $O(1)$ (these are the operations discussed above to compute the coefficients of a polynomial that results from the addition or multiplication of two other

polynomials). Essentially, this problem occurs because, as noted above, multiplying each row by any non-zero value does not change the equation the row defines. In the above procedure, each row, in some sense, accumulates extraneous multiplying factors. It will be desirable to remove these factors during the computation and thereby "simplify" each row.

One obvious idea would be to divide each row by the greatest common divisor (GCD) of all the elements (where here the elements are equivalence classes of polynomials, or equivalently elements of $\mathbb{Z}[\zeta_N]$). Despite the fact that we cannot, in general, divide elements in a ring, we can still certainly divide any element by one of its divisors. However, we still encounter difficultly in computing the GCD.

Over the integers, one can compute the GCD using Euclid's algorithm. The integers are a ring. Euclid's algorithm can be generalized to many other rings, which all called Euclidean domains. This includes the rings of integers of many number fields. To determine if Euclid's algorithm can be extended to the ring of integers of a particular cyclotomic field, we must first introduce a bit more terminology, see [4]. An *integral domain* is a ring with more than one element that has no zero divisors. An *ideal* A of a ring R is a subset of R such that if $a_1, a_2 \in A$ and $r \in R$, we have $a_1 + a_2 \in A$ and $ra_1 \in A$ (that is to say, it is closed under addition, and also under multiplication by any element in R). An ideal is thus clearly also a ring. We say an ideal A is *generated* by elements $g_1, \ldots, g_k \in R$ if A is the intersection of all ideals in R that contain $g_1, \ldots, g_k$. A *principal ideal* is an ideal generated by a single element. A *principal ideal domain* is an integral domain that only has principal ideals.

We can use the fact that a ring of integers of a number field is a Euclidean domain if and only if it is a principal ideal domain (PID) [4]. Unfortunately, as shown in [5], the set of N such that the ring of integers of the Nth cyclotomic field is a PID is finite.

In fact, for $N > 90$, the ring of integers of a cyclotomic field is never a PID [5]. This eliminates the possibility of using Euclid's algorithm.

In some sense, the failure to be a PID can be viewed as the failure for unique factorization to hold. For the integers, and more generally for any PID, we have the Fundamental Theorem of Arithmetic which states that every element can be uniquely factored into a product of powers of primes and a unit, where a unit is simply an invertible element (± 1 in $\mathbb{Z}$) and a prime is an indecomposable element (the usual primes in $\mathbb{Z}$). While we cannot uniquely factor elements of a non-PID ring of integers of a cyclotomic field, we can accomplish the same goal by factoring ideals.

We make use of Dedekind's Theorem [4] which states that in the ring of integers of any number field, we can uniquely factor any ideal into the product of powers of prime ideals. In fact, this holds for a more general class of rings called Dedekind rings, but this is not needed here. The idea will then be to factor out common terms from a row of the matrix by, for each element in the row, computing the principal ideal generated by that element, adding all the principal ideals together, and, if the result is a principal ideal, taking its generator. Every element of the row will be divisible by this generator, and so we can divide out that common factor. This algorithm is discussed in detail in [7].

This simplification procedure allows the above algorithm to be used as an alternative way to compute a relation between an unknown s_r and known s_j, $j \in J$. Both this algorithm and the original algorithm compute this relation exactly, and so the accuracy of both algorithms is identical.

Chapter 4

Classification

4.1 Fundamental Problem

The goal of a classification algorithm is to determine when each load in a collection of electrical loads turns on and off. The data used to make this determination is the aggregate current drawn by the collection of loads and the line voltage supplied to the loads. To begin to develop such an algorithm, a simpler fundamental problem will be examined first. Consider a black-box that contains a single, unknown electrical load drawn from a collection of electrical loads. The goal is to determine which load is in the black-box by examining the current drawn by that load when the unknown device is first turned on.

To be precise, let $L = \{l_1, \ldots, l_M\}$ denote a set of M electrical loads. A single load, l is selected from L according to some probability mass function $p_l(l_j) = Pr[l = l_j]$; that is to say, $p_l(l_j)$ denotes the probability that load l_j is selected. At this stage, $p_l(l_j)$ will be assumed to be known. Let $I = (i_0, \ldots, i_{N-1})$ denote the ordered N-tuple of

current samples drawn by the load l when it is turned on. These samples are collected uniformly in time, with n samples per period of the line voltage. It should be noted that I is a truncated version of the infinitely long vector of current samples that would be obtained by sampling the current waveform for all time. It will be assumed that that is some sufficiently long period of time such that ceasing sampling after this period of time will not cause the loss of any identifying information; that is to say, all relevant features of the current samples are contained in finitely many of the infinite collection of current samples.

The classification algorithm, A, takes input L and I and produces a prediction $\bar{l}$ of the identity of the load l. The goal is to maximize the probability that $\bar{l} = l$, which is, by definition, accomplished by setting

$$\bar{l} = argmax_{l_j \in L} Pr[l = l_j | I].$$

Using elementary probability, this can be rearranged as

$$\bar{l} = argmax_{l_j \in L} Pr[I | l = l_j] \frac{Pr[l = l_j]}{Pr[I]}. \tag{4.1}$$

In the above equation, $Pr[l = l_j]$ is simply the *apriori* probability that load l_j is in the box, which is given by $p_l(l_j)$. $Pr[I|l = l_j]$ is simply the probability of generating the current N-tuple I given the device l_j. It should be noted that, even for a single fixed load l_j, many different current N-tuples I could be measured due to noise and other factors (for example, a real electrical load might draw different current waveforms in warm and cold environments). $Pr[I]$ is the *apriori* probability that I will be the observed data,

which is given by

$$Pr[I] = \sum_j Pr[I|l = l_j]Pr[l = l_j].$$

Thus, in principle, the classification algorithm is quite trivial. Given a sufficiently detailed model of the electrical characteristics of every load, one could calculate $Pr[I|l = l_j]$ for any load $l_j \in L$ and, using (4.1), make the best possible prediction $\bar{l}$ of l. In practice, however, such a detailed model of the underlying physical properties of the loads is often unavailable. That is to say, while determining the physical properties of the devices may be possible, it is likely undesirable to perform a detailed analysis of a load before the algorithm could be used to identify that load. An alternative to having these detailed models, considered in the next section, is to construct a simplified model using observed data.

4.2 Device Modeling

As shown in the previous section, the development of a classification algorithm requires certain pieces of information about the collection of electrical loads. In particular, it is necessary to know the conditional probability distribution of current, $Pr[I|l = l_j]$, for each $l_j \in L$ and every possible current vector I, as well as the probabilities $p_l(l_j)$ that each load $l_j \in L$ will be the load in the black-box (which, up to this point, has been assumed to be known). While it is true that $Pr[I|l = l_j]$ could be calculated from sufficiently detailed *apriori* knowledge of the electrical characteristics of load l_j, it is, as noted in the previous section, often impractical to do so. This section will consider methods to estimate these unknown probabilities experimentally.

Consider a modified version of the black-box load problem stated above. In this

modified version of the problem, there are two phases: a training phase and a classification phase. In the training phase, data is gathered on the loads that will be used to estimate the unknown probabilities. In the classification phase, these estimated probabilities will be used in classification algorithm of the previous section to classify an unknown load in the black-box. To be precise, the training phase will consist of some large number K of trials. In each trial, a load $l \in L$ is selected according the probability distribution $p_l(l)$; the load is then turned on, and data is collected. Throughout the training phase, the loads are not operating in a black-box; that is to say, the identity of each load selected is known.

The data from the training phase can then be used to estimate the probabilities $Pr[I|l = l_j]$ for each load $l_j \in L$. A straightforward, but entirely impractical, way to do this would be simply count the number of times that any particular load l produced the current vector I, which will be denoted $K_{l,I}$, and the number of times that load l was selected, which will be denoted K_l, and then estimate $Pr[I|l = l_j]$ by the quantity $\frac{K_{l_j,I}}{K_l}$. This is completely impractical because, even though the number of possible current vectors is finite, it is extremely large. If each of the N samples of current is taken to b bits of precision, then there are 2^{bN} possible values of I. It would not be reasonable to even store 2^{bN} estimates for each of the M loads, much less actually produce all of the estimates.

A more practical approach is to assume that the distributions $Pr[I|l = l_j]$ have some simple functional form with only a few unknown parameters. A particularly useful form arises from assuming that each load l_j has a corresponding characteristic current vector I_j that it would draw under ideal circumstances. That is to say, in the complete absence of noise and measurement inaccuracy, if device l_j were turned on, the measured current would be I_j. Noise will be modeled as additive white Gaussian noise. Without

loss of generality, this noise can be assumed to be zero-mean, because if the noise had a non-zero mean, the mean could be estimated accurately over some long period of time and subtracted out. In this setting, only the characteristic current vectors $\{I_j\}$ for each of the loads $l_j \in L$, as well as the variance σ^2 of the noise (a Gaussian is completely determined by its mean and variance) needs to be estimated. This reduces the task of estimating $M2^{bN}$ parameters to estimating only $MN + 1$ parameters.

Additionally, it should be noted that, in this setting, it is no longer necessary to assume that the probability mass function $p_l(l_j)$ is known. This is because $p_l(l_j)$ can be approximated by $\frac{K_l}{K}$, the proportion of trials in the training phase in which load l_j appears.

4.3 Spectral Envelopes

Thus far, the problem of classification has only been considered using raw current as the source of data. This section will explore classification algorithms that instead make use of the Discrete Fourier Transform (DFT) of the raw current and the spectral envelope representation. For notational convenience, this chapter will use a slightly different definition of spectral envelopes than the original definition given in Chapter 2. In particular, a complex form of spectral envelopes is defined. All results in this chapter would apply equally well to spectral envelopes given by the original definition.

The DFT is defined as follows. Given a sequence of n complex numbers, $x_0, \ldots, x_{n-1}$, the DFT transforms this sequence into the n complex numbers $X_0, \ldots, X_{n-1}$ where

$$X_k = \sum_{j=0}^{n-1} x_j e^{-\frac{2\pi i}{n} kj}.$$

Similarly, the inverse Discrete Fourier Transform (IDFT), which expresses the original sequence in terms of the transformed sequence, is given by

$$x_j = \frac{1}{N} \sum_{k=0}^{n-1} X_k e^{\frac{2\pi i}{n} jk}.$$

Using this definition we can form spectral envelopes $S_0(t), \ldots, S_{n-1}(t)$ which are defined by

$$S_k(t) = e^{\frac{2\pi i}{n} tk} \sum_{j=0}^{n-1} i_{t+j} e^{-\frac{2\pi i}{n} jk}. \tag{4.2}$$

Thus, the kth spectral envelope at time t is simply the kth term in the DFT of the sequence $i_t, \ldots, i_{t+n}$, which are the samples of current over one line-cycle worth of time (there are n samples of current taken per line cycle of voltage), rotated by an appropriate factor. It should be noted that, because t can take any (integral) value, the sequence of n samples used to produce $S_k(t)$ at any particular time t will not necessarily start or end at zero-crossings of the line voltage.

The spectral envelopes are defined in this way so that they will correspond to meaningful physical quantities. For example, under the assumption of a stiff, harmonically pure line voltage, $\Im\{S_1(t)\}$ corresponds to real power, $\Re\{S_1(t)\}$ corresponds to reactive power, $\Im\{S_2(t)\}$ corresponds to power drawn at the second real harmonic, and so on.

In considering developing a classification algorithm that uses spectral envelope data, instead of raw current data, an immediate question is the accuracy of the resulting classifier. To frame this question properly, consider a slightly modified version of the black-box experiment from Section 1. Again, a randomly selected load $l \in L$ is placed in a black-box and data is recorded when this load is turned on. The change is that now,

instead of recording I, the sequence of N current samples, we record the power spectral envelopes.

Consider a classification algorithm E that takes as input L and $S_0(t), \ldots, S_{n-1}(t)$ for $t \in [0, N-n]$ (the input consists of the set L and values of each of the power spectral envelopes at each point in time) and outputs a prediction $\bar{l}$ of the identity of the load l with the goal of maximizing the probability that $\bar{l} = l$. Clearly, the best prediction that E can make is

$$\bar{l} = argmax_{l_j \in L} Pr[l = l_j | \{S_j(t) | j \in [0, n-1], t \in [0, N-n]\}].$$

The natural question to ask is which of the two predictions (made by each of the two algorithms A and E) has the higher probability of being correct. The answer to this question is that the predictions made by A and the predictions made by E have, in all cases, the same probability of being correct. This is shown by the following lemma that relies on the fact that both A and E are, by definition, optimal algorithms. That is to say, they produce the prediction that is most likely given their input (L and I in the case of A and L and $\{S_j(t) | j \in [0, n-1], t \in [0, N-n]\}$ in the case of E).

Lemma 2. *For algorithms A and E, as defined above, let $\bar{l}_A$ and $\bar{l}_E$ denote the predictions made by algorithms A and E, respectively. Then, $Pr[\bar{l}_A = l] = Pr[\bar{l}_E = l]$ always.*

Proof. Assume that there is some case for which the prediction of E is better (more likely to be correct) than the prediction of A. It is immediately clear that the input to E can be calculated from the input to A (by applying equation 2 above). Thus, it is possible to construct a third algorithm C which takes input L and I (the same input that A takes), produces $\{S_j(t) | j \in [0, n-1], t \in [0, N-n]\}$, runs E, and outputs the prediction made by E. By the assumption that E will, in some case, return a better prediction than A, it

follows that C would return a better prediction than A, which contradicts the optimality of A. This contradiction immediately implies that there is no case in which algorithm E could produce more accurate predictions than algorithm A.

Similarly, assume that there is some case for which the prediction of A is better than the prediction of E. Again, it is possible to produce the input to A from the input to E. This is due to the fact that the DFT is invertible; thus, given knowledge of $S_0(t), \ldots, S_{n-1}(t)$ for all time the vector I can be produced. Therefore, there exists an algorithm D which, on input L and $\{S_j(t) | j \in [0, n-1], t \in [0, N-n]\}$ could produce a better prediction than E by using A, which contradicts the optimality of algorithm E. This contradiction implies that there is also no case for which the prediction of E is better than the prediction of A. □

Given that algorithm E performs exactly as well as algorithm A, the natural next question to ask is whether there is any motivation in recording power spectral envelope data instead of raw current data. As will soon be demonstrated, a potential motivation is compression. To see this, consider the total amount of storage space needed for the input to algorithm A and the input to algorithm E. In both cases, the input includes L, and so this portion of the storage space requirement is the same in both cases. The other part of the input for algorithm A is the current vector I which consists of N samples, each of which are taken to a precision of b bits, and so storing the vector I requires Nb bits. Thus, in order for power spectral envelopes to be useful as a form of compression, they must require fewer than Nb bits to store.

Unfortunately, simply storing all of the spectral envelopes used as input to E would take considerably more space to store. Each of the n spectral envelopes must be recorded at $N - n + 1$ values of t; if $2b$ bits are used to record each of these spectral

envelope values (b bits for each of the real and imaginary parts of the spectral envelopes) then a total of $2bn(N-n+1)$ would be required.

However, there is still hope due to the tremendous redundancy in the spectral envelopes. That is to say, despite the fact that so many bits are used to store the spectral envelopes, their actual information content is much smaller. To see this, recall that the power spectral envelopes can be determined from the current vector I. Thus, it is never necessary to store more power spectral envelope values than are needed to uniquely determine I. This allows two types of redundancies to immediately be exploited. Firstly, since I is real (it consists of measured current values, and has no imaginary part), it must be the case that $X_j = X^*_{n-j}$, where $*$ denotes conjugation, (this is simply a property of the DFT) and so $S_j(t) = S_{n-j}(t)^* e^{\frac{2\pi i}{n}(2j-n)t}$. Thus, knowledge of $\{S_j(t)|j \in [0, \frac{n}{2}-1], t \in [0, N-n]\}$ would suffice. The second sort of redundancy arises from the fact that the DFT is invertible. Thus, given knowledge of the DFT of n sequential values of current (for example, from time t to $t+n-1$, as is used to construct $\{S_j(t)|j \in [0, n-1]\}$) would suffice to reconstruct those n values of current. Thus, it is only necessary to record the $S_j(t)$ at a spacing of n in time. Combining both of these redundancies, only half of the n spectral envelopes need to be recorded, and only at $\frac{N}{n}$ points in time. If each of the spectral envelopes are recorded to $2b$ bits of precision (b bits for the real part and b bits for the imaginary part) this will require exactly Nb bits, the same amount of storage space necessary to store the raw current values.

This has not accomplished any sort of compression because the same amount of storage space is needed to store spectral envelope values as was needed to store raw current values. In order to achieve actual compression, fewer spectral envelope values must be stored. It can be hoped that there is some additional redundancy in the data produced by real electrical loads that would allow fewer spectral envelope values to be

stored without loss of information.

As observed earlier, the power spectral envelopes correspond to real physical quantities (i.e. $\Im\{S_1(t)\}$ corresponds to real power). Thus, one could hope that it is the case that real electrical loads only draw power at a few harmonics, and so only a few of the spectral envelopes would be non-zero. If this were indeed the case, then a large amount of storage space could be saved by not storing envelopes with a constant 0 value. Unfortunately, as will soon be shown, even if only a few spectral envelopes are non-zero during steady-state portions of a waveform, many spectral envelopes will be non-zero during transient events.

Consider the following simple current vector shown below, in Fig. 4.1.

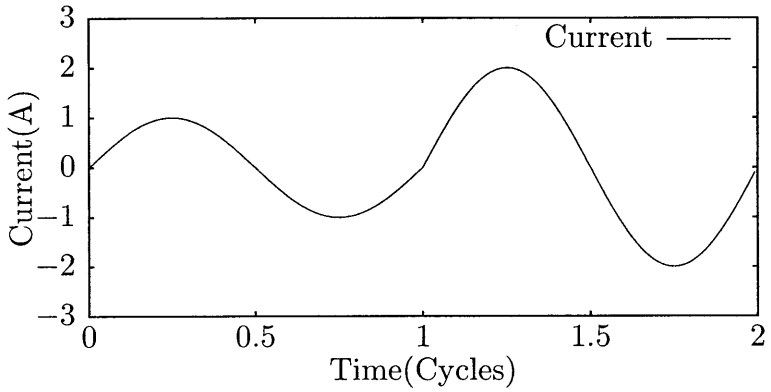
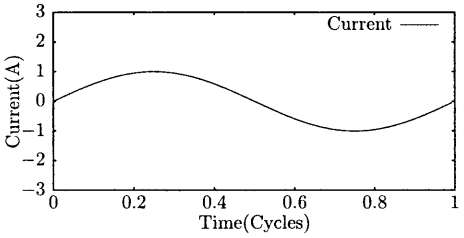


Figure 4.1: Current Vector.

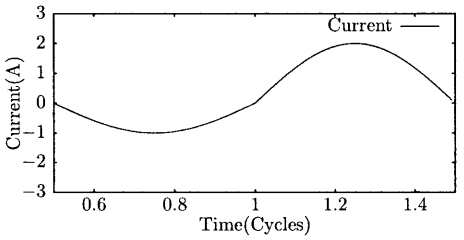
This current vector I consists of samples over two cycles of the line voltage. It is given by the equation

$$I(t) = \begin{cases} \sin(\frac{2\pi}{n}t) & t \in [0,1] \\ 2\sin(\frac{2\pi}{n}t) & t \in (1,2] \end{cases}$$

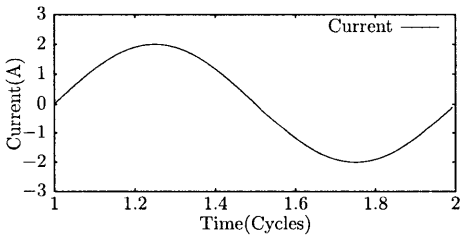
For clarity, Figure 2, below, shows views of this current waveform over three windows. The three windows are each of length one cycle (n data points) and start at an offset of $0, \frac{1}{2}$ and 1 cycles, respectively.



(a) First window.



(b) Second window.



(c) Third window.

Figure 4.2: Current Vector split into windows.

In this example, it can easily be shown that S_1, S_{n-1} and $\{S_{2k} | k \in [0, \frac{n}{2} - 1]\}$ are all not constantly equal to 0. In particular, $S_1(0) \neq 0$, $S_{n-1}(0) \neq 0$ and $S_{2k}(\frac{n}{2}) \neq 0, \forall k \in [0, \frac{n}{2} - 1]$. Despite the fact that this example is extremely specific, it illustrates an important general phenomenon. The current vector I is a piecewise function where, over each cycle, it consists of a single pure harmonic. Therefore, if we examine the power spectral envelopes at times given by the start of cycles of the line voltages (that is to say, at times when the power spectral envelope is calculated from samples all within the sample line cycle) then only two harmonics will be non-zero (only one of which needs to be recorded as each of these harmonics could be calculated from the other). However, if we examine the spectral envelopes at times given by the middle of line cycles (when half of the current values used to calculate the spectral envelopes come from two adjacent cycles) then we see that all harmonics whose parity differs from the single pure harmonic are non-zero.

Despite the fact that, as shown above, many spectral envelopes may not be identically equal to 0, this does not preclude the possibility that the vast majority of spectral envelopes are 0 at all points of interest. Recall that, as noted above, there is a great deal of redundancy in the spectral envelopes. In particular, it is only necessary to store spectral envelope data at a spacing of n in time. Thus, if it were the case that any spectral envelope satisfied $S_j(kn) = 0, \forall k \in [0, \frac{N}{n} - 1]$, then storage space could be saved by not recording spectral envelope S_j. For example, in the example above, the only spectral envelope that isn't equal to 0 at points in time that are multiples of n is S_1. Thus, this current vector could be stored (with no loss of information) by only storing S_1.

More generally, consider any current vector I that consists of full-period piecewise combinations of the harmonics $k_1, \ldots, k_p$. That is to say, over each cycle of the line voltage (n points of data) the current vector can be written as a linear combination of the complex

exponentials $e^{\frac{2\pi i}{n}k_1 t}, \ldots, e^{\frac{2\pi i}{n}k_p t}$. Then, clearly, at the points in time $\{kn | k \in [0, \frac{N}{n} - 1]\}$, the only non-zero spectral envelopes will be $S_{k_1}, \ldots, S_{k_p}$. Therefore, in such a case, storing the sufficient set of spectral envelopes (the set necessary to compute all spectral envelope values) only takes $\frac{p}{n}Nb$ bits to store, which could be much less than that Nb bits needed to store all raw current values if p is much smaller than n (that is to say, if only a small portion of the possible harmonics are actually used).

This same notion can be extended to a wider class of current vectors. Consider any current vector I that is a half-period piecewise combination of the harmonics $k_1, \ldots, k_p$. That is to say, over each half-cycle of the line voltage ($\frac{n}{2}$ points of data), I can be written as a linear combination of the complex exponentials $e^{\frac{2\pi i}{n}k_1 t}, \ldots, e^{\frac{2\pi i}{n}k_p t}$. Then, as shown in the following lemma, there are many cases in which recording the spectral envelopes $S_{k_1}, \ldots, S_{k_p}$ only at the points $\{k\frac{n}{2} | k \in [0, \frac{2(N-1)}{n}]\}$ (plus a negligible amount of side information) would suffice to completely determine all spectral envelope values. This would take only $2\frac{p}{n}Nb$ bits (plus a negligible amount to store side information), which again could be much less than the Nb bits needed to directly store all raw current values.

Lemma 3. *Let $k_1, \ldots, k_p \in [0, n-1]$ be a collection of distinct harmonics such that the matrix $B = (b_{l,j})$ where $b_{l,j} = \sum_{h=0}^{\frac{n}{2}-1} e^{\frac{2\pi i}{n}(k_j - k_l)h}$ is invertible. Let $I = (i_0, \ldots, i_{N-1}) \in \mathbb{R}^N$ be any current vector that consists of half-period piecewise combinations of those harmonics. Then, given $i_0, \ldots, i_{\frac{n}{2}-1}$ and $\{S_k(t) | k \in \{k_1, \ldots, k_p\}, t \in \{\frac{mn}{2} | m \in [0, \frac{2(N-1)}{n}]\}\}$ it is possibly to completely determine I uniquely.*

Proof. This can be shown by induction. As the base case, note that $i_0, \ldots, i_{\frac{n}{2}-1}$ are uniquely specified. Then, for each $t \in \{\frac{mn}{2} | m \in [0, \frac{2(N-1)}{n}]\}$, if $i_t, \ldots, i_{t+\frac{n}{2}-1}$ are uniquely specified then $i_{t+\frac{n}{2}}, \ldots, i_{t+n-1}$ can be uniquely determined as follows.

Let $m_l = \sum_{j=t}^{t+\frac{n}{2}-1} i_j e^{-\frac{2\pi i}{n}lj}$. Each of the m_l are determined, uniquely, by the

already known values of I. Let $a_1, \ldots, a_p \in \mathbb{C}$, $f(x) = \sum_{j=1}^{p} a_j e^{\frac{2\pi i}{n} k_j x}$. Then $i_j = f(j - t - \frac{n}{2}), \forall j \in \{t - \frac{n}{2}, t + n - 1\}$ for some setting of $a_1, \ldots, a_p$ because I consists of half-period piecewise combinations of the harmonics $k_1, \ldots, k_p$. Then,

$$
\begin{aligned}
S_{k_l}(t) &= \sum_{j=t}^{t+n-1} i_j e^{-\frac{2\pi i}{n} k_l j} \\
&= \sum_{j=t}^{t+\frac{n}{2}-1} i_j e^{-\frac{2\pi i}{n} k_l j} + \sum_{t=t+\frac{n}{2}}^{t+n-1} i_j e^{-\frac{2\pi i}{n} k_l j} \\
&= m_l + \sum_{h=0}^{\frac{n}{2}-1} f(h) e^{-\frac{2\pi i}{n} k_l (h+\frac{n}{2})} \\
&= m_1 + \sum_{h=0}^{\frac{n}{2}-1} \sum_{j=1}^{p} a_j e^{\frac{2\pi i}{n} k_j h} e^{-\frac{2\pi i}{n} k_l h} e^{-\pi i k_l} \\
&= m_l + (-1)^{k_l} \sum_{h=0}^{\frac{n}{2}-1} \sum_{j=1}^{p} a_j e^{\frac{2\pi i}{n} (k_j - k_l) h} \\
&= m_l + (-1)^{k_l} \sum_{j=1}^{p} a_j \sum_{h=0}^{\frac{n}{2}-1} e^{\frac{2\pi i}{n} (k_j - k_l) h} \\
&= m_l + (-1)^{k_l} \sum_{j=1}^{p} a_j b_{l,j},
\end{aligned}
$$

where $b_{l,j} = \sum_{h=0}^{\frac{n}{2}-1} e^{\frac{2\pi i}{n}(k_j - k_l)h}$. Let $A = [a_1, \ldots, a_p]^T$ be the $1 \times P$ column vector of the a_js, let $B = (b_{l,j})$ be the $p \times p$ matrix of the $b_{l,j}$s, and let $C = [(-1)^{k_1} S_{k_1} - m_1, \ldots, (-1)^{k_p} S_{k_p} - m_p]^T$ be a $1 \times P$ column vector. Then, the above becomes $BA = C$ which has the unique solution $A = B^{-1}C$ exactly when B is invertible. Since B is invertible by assumption, this immediately implies that $i_{t+\frac{n}{2}}, \ldots, i_{t+n-1}$ are uniquely determined. Inducting on t completes the proof. □

The applicability of this lemma depends on the invertibility of this matrix B. A necessary and sufficient condition for the invertibility of B is shown in the following lemma.

Lemma 4. *Let $k_1, \ldots, k_p \in [0, n-1]$ be a collection of distinct harmonics, and let $B = (b_{i,j})$ be the $p \times p$ matrix with entries given by $b_{l,j} = \sum_{h=0}^{\frac{n}{2}-1} e^{\frac{2\pi i}{n}(k_j - k_l)h}$. Then B is invertible if and only if $p \leq \frac{n}{2}$.*

Proof. Let $y_0, \ldots, y_{n-1}$ be n element vectors where the h^{th} element of vector y_j is given by

$$y_j[h] = e^{\frac{2\pi i}{n} jh}.$$

Thus, the y_j are simply the DFT basis functions. Let $x_1, \ldots, x_p$ denote n element vectors which are given by the p elements of the set $\{y_j\}$ that correspond to elements in $\{k_j\}$. That is to say $x_j = y_{k_j}$. Let $\overline{x_1}, \ldots, \overline{x_p}$ be n element vectors that are defined by

$$\overline{x_j}[h] = \begin{cases} x_j[h] & h < \frac{n}{2} \\ 0 & \text{otherwise} \end{cases}.$$

Similarly, let

$$\overline{y_j}[h] = \begin{cases} y_j[h] & h < \frac{n}{2} \\ 0 & \text{otherwise} \end{cases}.$$

Recall that the Gram matrix of a set of n element vectors $\{v_1, \ldots, v_p\}$ is the $p \times p$ matrix given by $G = (g_{i,j})$ where

$$g_{i,j} = \langle v_i, v_j \rangle.$$

Furthermore, recall that G is invertible if and only if the vectors $\{v_1, \ldots, v_p\}$ are linearly independent. Clearly, B is the Gram matrix of $\{\overline{x_1}, \ldots, \overline{x_p}\}$. Thus, B is invertible if and only if $\{\overline{x_1}, \ldots, \overline{x_p}\}$ is a linearly independent set.

Next, notice that $y_0, \ldots, y_{n-1}$ is an orthogonal basis of the space $\mathbb{C}^n$. Moreover, $\overline{y_0}, \ldots, \overline{y_{n-1}}$ are the projection of $y_0, \ldots, y_{n-1}$ onto the space $((\mathbb{C}^{\frac{n}{2}} \oplus 0^{\frac{n}{2}}) \cong \mathbb{C}^{\frac{n}{2}}$. Clearly, $\overline{y_0}, \ldots, \overline{y_{n-1}}$ span the $\frac{n}{2}$ dimensional space $((\mathbb{C}^{\frac{n}{2}} \oplus 0^{\frac{n}{2}})$. Thus, any subset of $\frac{n}{2}$ elements of the set $\{\overline{y_0}, \ldots, \overline{y_{n-1}}\}$ must be a basis of $((\mathbb{C}^{\frac{n}{2}} \oplus 0^{\frac{n}{2}})$, which immediately implies that any subset of $\{\overline{y_0}, \ldots, \overline{y_{n-1}}\}$ of size at most $\frac{n}{2}$ is linearly independent, and of size at least $\frac{n}{2}$ is not linearly independent.

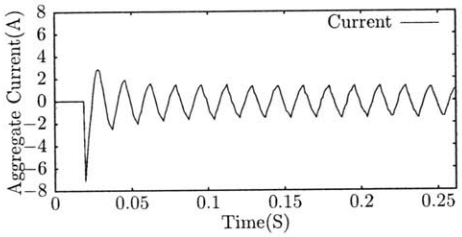
By construction, $\{\overline{x_1}, \ldots, \overline{x_p}\} \subset \{\overline{y_0}, \ldots, \overline{y_{n-1}}\}$. Therefore, $\{\overline{x_1}, \ldots, \overline{x_p}\}$ is linearly independent if and only if $p \leq \frac{n}{2}$.

□

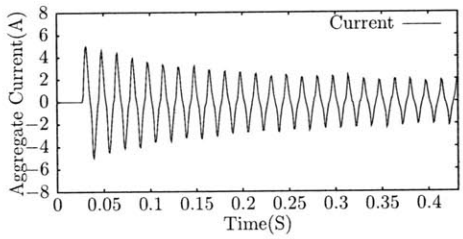
Based on observations of a variety of loads, [1], it appears that many real electrical loads consist (approximately) of half-period piecewise combinations of a small number of harmonics. Thus, it is reasonable to expect that the above assumptions might actually be valid in practice. Figure 3, below, shows the raw current drawn by a variety of loads which have this property.

This section concludes by considering an algorithm P which takes as input L and any sufficient set of spectral envelope values $\{S_j(t)\}$. That is to say, the set of spectral envelope values is sufficient to uniquely determine the characteristic current vector I_j for any load $l_j \in L$. P makes its prediction by the applying the same maximum likelihood rule used by algorithms A and E. That is to say, it makes the prediction

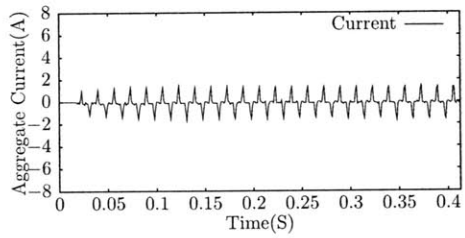
$$\bar{l} = argmax_{l_j \in L} Pr[l = l_j | \{S_j(t)\}].$$



(a) Incandescent Light Bulb.



(b) Motor.



(c) Computer Power Supply.

Figure 4.3: Turn-on Transients. This figure illustrates turn on transients for a variety of electrical loads. These three plots show a turn on transient for an incandescent light bulb, a motor, and a computer power supply, respectively.

The following lemma shows that such an algorithm will perform exactly as well as the algorithm E, discussed earlier, whose input includes all spectral envelope values.

Lemma 5. *For algorithms P and E, as defined above, let $\bar{l}_P$ and $\bar{l}_E$ denote the predictions made by algorithms P and E, respectively. Then, $Pr[\bar{l}_P = l] = Pr[\bar{l}_E = l]$ always.*

Proof. The proof of them lemma is analogous to the proof of Lemma 1. Notice that, by construction, both E and P are optimal algorithms in the sense that they make the most likely prediction given their input. Moreover, notice that the input to P could be constructed from the input to E (because it is a subset of that input) and, furthermore, that the input to E could be constructed from the input to P (because the input to P is sufficient to uniquely determine I and therefore to uniquely determine all spectral envelopes).

Thus, by the logic of the proof of Lemma 1, if E (resp. P) performed better than P (resp. E) in any case, this would contradict the optimality of P (resp. E) because it would allow some other algorithm C to be formed which, on the same input as P (resp. E) would predict more accurately than E (resp. P) by using P (resp. E) as a subroutine. This contradiction immediately implies that $Pr[\bar{l}_P = l] = Pr[\bar{l}_E = l]$ always. □

4.4 EM Algorithm

The previous section discussed the theoretical applicability of using spectral envelope data for the purpose of classification. As was shown, a maximum-likelihood classifier using a sufficient set of spectral data is exactly as accurate at a maximum-likelihood classifier operating on the raw current data. This section will consider a practical way to perform classification using the EM algorithm [8].

The fundamental goal of the EM algorithm is to be able to take a large collection of data, where each piece of data is a set of spectral envelope data for a single turn-on event, and split that data into a collection of clusters, where, ideally, each cluster will contain all of the turn-on events for a single load. To be precise, the EM algorithm produces a maximum likelihood estimation of a set of unknown parameters given incomplete data. In this setting, the incomplete data is the collection of spectral envelope data obtained from many turn-on events. This data is incomplete because the identity of the load represented in each turn-on event is initially unknown (it is the goal of this classification algorithm to determine this identity). In the following, the "label" of a piece of (spectral envelope) data will refer to the identity of the load that produced that data. Data which has a label will be referred to as "labeled" and data without a label will be referred to as "unlabeled". The parameters that are being estimated by the algorithm are the parameters that determine the clusters.

For the sake of clarity, before providing a detailed description of the operation of the EM algorithm, a simple example will be considered. In this example, each cluster will be given by a multidimensional Gaussian in spectral envelope space. That is to say, if p significant spectral envelopes are recorded, at each of h different points in time, then the space of interest will be isomorphic to $\mathbb{C}^{ph}$ and each cluster will be given by a ph dimensional complex Gaussian. Thus, each cluster is represented by a probability density function where the value of this function at any point in (spectral envelope) space signifies the density of probability that the device represented by that cluster would produce the spectral envelope data represented by that point. Each Gaussian cluster will be assumed to be spherically symmetric; that is to say, its covariance matrix is given by $\sigma^2 I_{ph}$, where I_{ph} is the $ph \times ph$ identity matrix and σ^2 is a variance. Furthermore, in this example, all clusters will be assumed to have the same covariance matrix. Thus, the unknown

parameters that specify the M clusters (one cluster for each of the M electrical loads) are the means of each cluster $\mu_1, \ldots, \mu_M$ and the single common variance σ^2. It should be noted that these parameters really do suffice to completely specify the clusters because a Gaussian is determined by its mean and covariance matrix. In addition to the unknown parameters that specify the clusters, the *apriori* probability distribution of load selection is also unknown. As discussed in Section 1, $p_l(l_j)$ is the probability that load l_j is the load in the "black-box." This distribution is determined by $M-1$ parameters (there are M loads, and the sum of the probabilities over all loads must be 1). The EM algorithm will attempt to estimate the $2M$ unknown parameters ($M+1$ that specify the clusters, $M-1$ that specify the *apriori* distribution) from the data by finding the maximum likelihood setting of these parameters.

Before proceeding further, it is worth noting that, while many specific assumptions were made in the above example about the structure of the clusters, these assumptions are actually quite realistic, in certain settings. For example, consider raw current measurements that are corrupted with additive white Gaussian noise (AWGN). It can easily be shown that every spectral envelope will also be corrupted by Gaussian noise, and, moreover, that the variance of the noise in each spectral envelope will be equal (because the noise is white). Thus, in the AWGN case, the assumption that the clusters will be spherically symmetric Gaussians with identical covariance matrices would be exactly correct.

To describe the EM algorithm precisely, a bit of notation needs to be introduced. Let θ be a vector which specifies the parameters of the clusters (in the above example, θ consists of the means and common variance of the clusters as well as the probability distribution that each cluster). Let x be the vector of unlabeled data, which consists of spectral envelope values at each turn-on event. To be precise, if the data set includes

data from r turn-on events, then x is an r element vector where the j^{th} element of this vector is the collection of spectral envelope data during the j^{th} turn-on event. Thus, each element of x isn't a single number, but rather an entire collection of data. The vector x is a single sample value from the random vector X. Let Z denote the random vector of labels (each element is the identity of the load that produced a particular collection of spectral envelopes, stored in part of x, when turned on) and z denote a particular setting of labels (z is a particular sample value from Z). Let $L(\theta, x, z)$ denote the likelihood function, which specifies the likelihood that clusters with parameters θ would correspond to data x and z.

The algorithm operates is a series of phases, where each phase produces a new (and hopefully better) estimate of θ; let $\theta^{(t)}$ denote the estimate of θ produces in round t. Each phase consists of two steps: the expectation step (E-step) and the maximization step (M-step).

In the E-step, the expected value of the log of likelihood function is calculated (where expectation is performed with respect to the conditional distribution of Z given θ and x). This expected value, in step t, is denoted by $Q(\theta|\theta^{(t)})$ and is given by

$$Q(\theta|\theta^{(t)}) = E_{Z|x,\theta^{(t)}} \ln L(\theta, x, Z). \tag{4.3}$$

In the M-step, a new estimate of parameters, $\theta^{(t+1)}$, is produced by selecting the parameter θ which maximizes the quantity calculated in the E-step. Thus,

$$\theta^{(t+1)} = argmax_{\theta} Q(\theta|\theta^{(t)}). \tag{4.4}$$

To apply the EM algorithm to classifying loads on the basis of spectral envelope

data, the assumptions of the above example will be made, with the exception of the fact that no assumptions will be made about the covariance matrices of each Gaussian cluster. That is to say, different clusters may have different covariance matrices and each cluster may or may not be spherically symmetric. Thus, θ is the $3M$ element parameter vector given by

$$\theta = (\mu_1, \ldots, \mu_M, \Sigma_1, \ldots, \Sigma_M, p_1, \ldots, p_M)$$

where $\mu_1, \ldots, \mu_M$ are the means of the M clusters, $\Sigma_1, \ldots, \Sigma_M$ are the covariance matrices of each cluster, and $p_1, \ldots, p_M$ are the *apriori* probabilities that loads $l_1, \ldots, l_M$, respectively, are turned on. These *apriori* probabilities have the constraint $\sum_{j=1}^{M} p_j = 1$.

The likelihood function, $L(\theta, x, z)$, is given by

$$L(\theta, x, z) = \prod_{i=1}^{R} p_{z_i} f_{z_i}(x_i), \tag{4.5}$$

where $f_j(x)$ denotes the probability density function of the j^{th} cluster, x_i denotes the ith piece of data, and R denotes the number of pieces of data. This cluster is a multivariate Gaussian with mean μ_j and covariance Σ_j. Recall that the probability density function $f(x)$ of a multivariate Gaussian with mean μ and covariance matrix Σ is given by

$$f_j(x) = \frac{1}{\sqrt{(2\pi)^N \det(\Sigma_j)}} e^{-\frac{1}{2}(x-\mu_j)^T \Sigma_j^{-1}(x-\mu_j)}. \tag{4.6}$$

Therefore, by substituting (4.6) into (4.5), and rearranging, the likelihood function is given by

$$\begin{aligned} L(\theta, x, z) &= \prod_{i=1}^{R} \frac{p_{z_i}}{\sqrt{(2\pi)^N \det(\Sigma_{z_i})}} e^{-\frac{1}{2}(x_i-\mu_{z_i})^T \Sigma_j^{-1}(x_i-\mu_{z_i})} \\ &= \prod_{i=1}^{R} e^{\ln\left(\frac{p_{z_i}}{\sqrt{(2\pi)^N \det(\Sigma_{z_i})}}\right)} e^{-\frac{1}{2}(x_i-\mu_{z_i})^T \Sigma_{z_i}^{-1}(x_i-\mu_{z_i})} \end{aligned}$$

$$= \exp(\sum_{i=1}^{R}(\ln(p_{z_i}) - \frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln(\det(\Sigma_{z_i}))) - \frac{1}{2}(x_i - \mu_{z_i})^T\Sigma_{z_i}^{-1}(x_i - \mu_{z_i})). \quad (4.7)$$

The likelihood function is expressed in exponential family form for convenience in later calculation.

The goal of the E-step is to calculate $Q(\theta|\theta^{(t)}) = E_{Z|x,\theta^{(t)}} \ln L(\theta, x, Z)$. To do this, we must first calculate $P(Z_i = l_j|X_i = x_i)$. By Bayes' Theorem and the definition of conditional probability, this is given by

$$P(Z_i = l_j|X_i = x_i) = \frac{P(X_i = x_i|Z_i = l_j)P(Z_i = l_j)}{P(X_i = x_i)}$$

$$= \frac{P(X_i = x_i|Z_i = l_j)P(Z_i = l_j)}{\sum_{k=1}^{R} P(X_i = x_i|Z_i = l_j)P(Z_i = l_j)}$$

$$= \frac{f_j(x_i)p_j}{\sum_{k=1}^{R} f_k(x_i)p_k}. \quad (4.8)$$

Using this equation for $P(Z_i = l_j|X_i = x_i)$, and (4.3), we can immediately write a closed form equation for the calculation performed in the E-step.

$$Q(\theta|\theta^{(t)}) = E_{Z|x,\theta^{(t)}} \ln L(\theta, x, Z)$$

$$= E_{Z|x,\theta^{(t)}} \ln(\exp(\sum_{i=1}^{R}(\ln(p_{z_i}) - \frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln(\det(\Sigma_{z_i}))) - \frac{1}{2}(x_i - \mu_{z_i})^T\Sigma_{z_i}^{-1}(x_i - \mu_{z_i})))$$

$$= \sum_{j=1}^{M} P(Z_i = l_j|X_i = x_i) \sum_{i=1}^{R}(\ln(p_{z_i}) - \frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln(\det(\Sigma_{z_i}))) - \frac{1}{2}(x_i - \mu_{z_i})^T\Sigma_{z_i}^{-1}(x_i - \mu_{z_i})$$

$$= \sum_{j=1}^{M}\sum_{i=1}^{R} \frac{f_j(x_i)p_j}{\sum_{k=1}^{R} f_k(x_i)p_k}(\ln(p_{z_i}) - \frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln(\det(\Sigma_{z_i})) - \frac{1}{2}(x_i - \mu_{z_i})^T\Sigma_{z_i}^{-1}(x_i - \mu_{z_i})). \quad (4.9)$$

In the M-step, we choose a set of parameters $\theta^{(t)}$ to maximize the quantity

$Q(\theta|\theta^{(t)})$ determined in the E-step. Notice that $Q(\theta|\theta^{(t)})$ has a particularly simple form. None of the parameters $p_1, \ldots, p_M$ share a term with any of the other parameters $\mu_1, \ldots, \mu_M, \Sigma_1, \ldots, \Sigma^M$. Thus, the values of $p_1, \ldots, p_M$ that maximize $Q(\theta|\theta^{(t)})$ can be determined independently of $\mu_1, \ldots, \mu_M, \Sigma_1, \ldots, \Sigma_M$. Let $(p_1^{(t)}, \ldots, p_M^{(t)})$ denote the vector composed of the probabilities $p_1, \ldots, p_M$ produced in the preceding M-step. Then $(p_1^{(t)}, \ldots, p_M^{(t)})$, the estimates produced in the current M step, are given by

$$(p_1^{(t)}, \ldots, p_M^{(t)}) = argmax_{(p_1,\ldots,p_M)} \sum_{i=1}^{M} \sum_{j=1}^{r} \frac{f_j(x_i) p_j^{(t)}}{\sum_{k=1}^{r} f_k(x_i) p_k^{(t)}} \ln(p_i^{(t+1)}).$$

Notice that this has exactly the same form as the maximum-likelihood estimator for the binomial distribution. Thus,

$$p_i^{(t+1)} = \frac{1}{r} \sum_{j=1}^{r} \frac{f_j(x_i) p_j^{(t)}}{\sum_{k=1}^{r} f_k(x_i) p_k^{(t)}}.$$

Similarly, the pairs of parameters $(\mu_1, \Sigma_1), \ldots, (\mu_M, \Sigma_M)$ also appear in separate terms from one another and from the parameters $p_1, \ldots, p_M$. Therefore, the values of the parameters (μ_j, Σ_j) that maximize $Q(\theta|\theta^{(t)})$ can be determined independently of all other parameters. Thus,

$$(\mu_j^{(t+1)}, \Sigma_j^{(t+1)}) = argmax_{\mu_j,\Sigma_j} \sum_{i=1}^{R} \frac{f_j(x_i) p_j}{\sum_{k=1}^{R} f_k(x_i) p_k} (\frac{1}{2} \ln(\det(\Sigma_j)) - \frac{1}{2}(x_i - \mu_j)^T \Sigma_j^{-1} (x_i - \mu_j)).$$

Notice that this has exactly the same form as the maximum-likelihood estimator for a Gaussian distribution. Thus

$$\mu_j^{(t+1)} = \frac{\sum_{i=1}^{R} \frac{f_j(x_i) p_j x_i}{\sum_{k=1}^{R} f_k(x_i) p_k}}{\sum_{i=1}^{R} \frac{f_j(x_i) p_j}{\sum_{k=1}^{R} f_k(x_i) p_k}} \tag{4.10}$$

and

$$\Sigma_j^{(t+1)} = \frac{\sum_{i=1}^{R} \frac{f_j(x_i)p_j||x_i-\mu_j^{(t+1)}||^2}{\sum_{k=1}^{R} f_k(x_i)p_k}}{\sum_{i=1}^{R} \frac{f_j(x_i)p_j}{\sum_{k=1}^{R} f_k(x_i)p_k}}. \tag{4.11}$$

Chapter 5

An FPGA-based Spectral Envelope Preprocessor

5.1 Background

Power electronics and power electronic controls are proliferating in consumer electronics. There is an increasing expectation that advanced power conditioning electronics will play a role in managing and coordinating power consumption not simply for a particular load, e.g., a variable speed drive in an air conditioning plant, but also in response to the dynamic needs and capability of the utility system. Loads that can respond not only to their own tasking but also to the needs of the utility are implicit in many visions of a "smart grid."

There is a need for flexible, inexpensive metering technologies that can be deployed in many different monitoring scenarios. Individual loads may be expected to compute information about their power consumption. They may also be expected to communicate

information about their power consumption through wired or wireless means. Switch gear like circuit breaker panels may eventually be expected to provided detailed submetering information for different loads on different breakers or clusters of breakers and controls. New utility meters will need to communicate bidirectionally, and may need to compute parameters of power flow not commonly assessed by most current meters.

Appropriate sensing hardware and information delivery systems remain a chief bottleneck for many applications. Both vendors and consumers will likely find innumerable ways to mine information if made available in a useful form. However, metering hardware and access to metered information will likely limit the implementation of new electric energy conservation strategies in the near future. The U.S. Department of Energy has identified "sensing and measurement" as one of the "five fundamental technologies" essential for driving the creation of a "Smart Grid" [13]. Consumers will need "simple, accessible..., rich, useful information" to help manage their electrical consumption without interference in their lives [13].

Digital technology has been in use for over 20 years for measuring and metering power flow. A few examples from an enormous array of metering and measurement approaches for monitoring power can be found in [14], [15], and the references in these documents. Digital power monitoring has also made its way to the "plug" and "power strip" level, e.g., see [17]. Many different schemes for storing or communicating information are still under exploration – see [16] and its references for example. Most of these solutions deploy computation hardware that is either substantially complicated in both hardware and firmware, e.g., [14] where a DSP and a micro-controller work together to coordinate computation of real, reactive, and apparent power, or where fully integrated custom chips are specifically developed for a particular application.

The "spectral envelope" representation of observed current and voltage signals

used in the non-intrusive load monitor or NILM [1] can be a very flexible basis for computing and tracking all sorts of useful metrics about power consumption. Spectral envelopes estimate real and reactive power consumption and harmonic contents. Also, even for waveforms with substantial high frequency content, the frequency content of the spectral envelopes can be made relatively band-limited. Spectral envelopes are often a natural way to "compress" useful data about load current and power consumption, easing communication requirements.

This chapter describes an integer-arithmetic implementation of a spectral envelope preprocessor for an inexpensive FPGA. The spectral envelope FPGA coordinates data acquisition and computes spectral envelopes without the need for floating point computation. Hence, the FPGA can be used in a two-IC suite (with an analog-to-digital converter), to inexpensively acquire load consumption data. This data minimizes the need for "downstream" computation later in the signal processing workflow. Of course, further computation can be used to track, trend, price, or control energy consumption. The FPGA can directly control communication as well, providing wired or wireless access, or storage on flash memory or other media. The spectral envelope FPGA is a cost-effective building block that can be used to enable a huge array of power monitoring and control applications, ranging from the individual load up through the breaker panel or utility service entry level and beyond. It can be used to provide necessary power consumption information for coordinating power electronic controls.

This chapter describes the design of this key building block and shows results from a prototype. The next section describes the approach for using spectral envelopes for load identification and how this data is computed by the preprocessor. The following section presents the FPGA-based spectral envelope preprocessor and the techniques used to implement the preprocessor cost-effectively. Finally, the performance of prototype

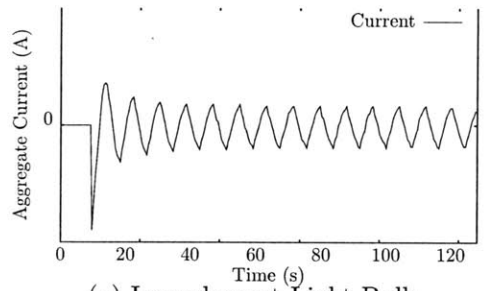
hardware is examined and further enhancements for expanded monitoring applications are described.

5.2 Utility of Spectral Envelopes

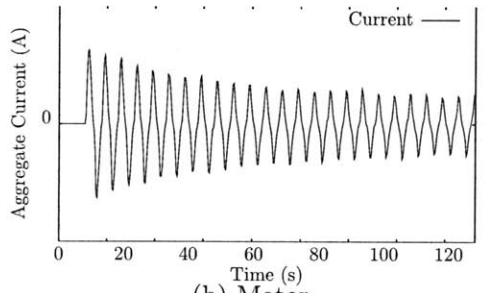
Typical turn-on current transients are shown in Fig. 5.1 for an incandescent lamp, a universal motor (as in a vacuum cleaner or hand tool), and a personal computer. Dynamic changes in the power and harmonic consumption of a load, e.g., during turn-on or turn-off transients, can serve as a fingerprint for identifying load operation [1]. For example, an observed turn-on transient or exemplar from a training observation produced by one of a collection of loads can be used to identify the load in an aggregate current measurement. An analogous procedure can be performed using turn-off transients. All that is needed, in principle, to determine the operating schedule of a collection of loads is to record the aggregate current drawn by those loads and then match each observed transient to the turn-on or turn-off fingerprint of a particular load in the collection.

Direct examination of current waveforms may not be practical for many stages of some applications, including many components in energy scorekeeping, monitoring, or conservation systems. Direct operations on the current waveform require sample rates adequate to capture the highest harmonic content of interest [1]. In some metering, monitoring, and control applications it is more practical to either store data for a period of time and examine it later, or transmit data to another location for interpretation and control. In either of these cases, it is convenient to have a useful representation of the data that avoids excessive storage or communication bandwidth requirements.

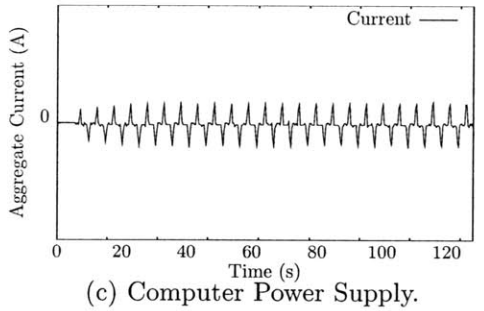
Spectral envelopes provide a useful separation between data collection and analysis. They permit a small, inexpensive system with low processing power to collect data



(a) Incandescent Light Bulb.



(b) Motor.



(c) Computer Power Supply.

Figure 5.1: Turn-on transients for an incandescent light bulb, a motor, and a computer power supply, respectively.

continuously. A system with larger available processing power, potentially physically remote from the data collection front end, can either review a storage device at a later time or continuously process a relatively low bandwidth information stream over a convenient communication channel, wired or wireless.

The spectral envelopes of current are defined at each line-locked period of the service voltage. If $i[n]$ represents the samples of current, and there are N samples taken per cycle, then the spectral envelopes of current are given by $a_0[m], \ldots, a_{N-1}[m]$ and $b_0[m], \ldots, b_{N-1}[m]$, where m indexes the period. These spectral envelopes are defined as

$$a_j[m] = \sum_{k=0}^{N-1} i[mN+k] \cdot sin(\frac{2\pi k}{N}) \tag{5.1}$$

and similarly,

$$b_j[m] = \sum_{k=0}^{N-1} i[mN+k] \cdot cos(\frac{2\pi k}{N}). \tag{5.2}$$

Spectral envelopes can be expressed in terms of the Discrete Fourier Transform (DFT) [2]. The DFT transforms a sequence of N complex numbers, $x[0], \ldots, x[N-1]$, to another sequence of N complex numbers $X[0], \ldots, X[N-1]$ by

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-\frac{2\pi j}{N} kn}$$

where here j, rather than i is used to represent $\sqrt{-1}$, to avoid confusion with current. The inverse of this transformation, the IDFT, is given by

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{\frac{2\pi j}{N} kn}.$$

Thus, the spectral envelope values are simply given by the real and imaginary parts of

the DFT of current.

Given the DFT coefficients over one period of the service voltage, it is possible to exactly reconstruct or preserve all of the information in the raw current samples over that period. Of course, simply recording all of the DFT coefficients will not reduce the data handling requirements – if there are N samples of current, the DFT will transform these N samples to N DFT coefficients. While it may appear that storing the DFT coefficients would take twice as much space as storing the raw current samples, because they are complex numbers, it should be noted that the DFT will be symmetric (because the raw current samples are real), so only $\frac{N}{2}$ of the N complex numbers need to be stored; thus, the storage requirement for storing all meaningful DFT coefficients is the same as storing all raw current samples (when both are stored to the same level of precision). However, as observed in [10], in situations where the significant or relevant current drawn by an electrical load consists predominantly of the fundamental frequency (the frequency of the service voltage, for example, 60 Hz) and a small collection of the line frequency harmonics (such as the 2^{nd}, 3^{rd}, 5^{th}), it is reasonable to record, for each period of the service voltage, only a few DFT coefficients. These relatively few DFT coefficients can be used to reconstruct the original current samples with a relatively small error. Also, the time varying values of the DFT coefficients themselves can be used directly as fingerprint signatures for the loads, or to track important quantities associated with load operation with reasonable accuracy.

Because only a few DFT coefficients may be needed to accurately represent the current waveforms, this "spectral" approach to the representation of the waveforms serves as a form of compression. As a concrete example, consider current and voltage samples that are collected at a 7.68 KHz sample rate. The sampling rate must be sufficiently high in order to provide adequate anti-aliasing without filtering effects and to provide

adequate detection of voltage zero crossings to enable line-locked data collection. This corresponds to 128 samples per 60 Hz line-cycle ($N = 128$), and so 64 meaningful complex DFT coefficients need to be stored to perfectly reconstruct arbitrary current samples. However, for many applications, including load monitoring for diagnostics, only a limited number of DFT coefficients need be stored. In the prototype system discussed here, just 4 coefficients, or 6.25% of the full set of already compact harmonically-related DFT coefficients, were needed.

Of course, other reductions of the data could be applied, e.g., simply recording average aggregate real power once per second (where the average is taken over each second interval), leads to further compression. Such data would not reflect the detailed short term variations that would occur in real power, nor would it reflect any of the behavior of the higher harmonics. Time-varying DFT coefficients or spectral envelopes strike a balance between the need to store or transmit as little data as possible and the need to maintain sufficiently detailed data to be able to accurately perform load monitoring of interest.

Spectral envelopes can be directly interpreted as other meaningful physical quantities under some conditions. In situations as illustrated in Fig. 5.2 where the utility voltage is relatively harmonic free and "stiff" (with constant peak amplitude), the real part of the fundamental frequency spectral envelope or DFT coefficient of the current waveform scales to "real power" in steady-state. The imaginary part scales to reactive power. If the voltage is not stiff or harmonically pure, then it is still possible to accurately estimate real and reactive power by also storing the same set of DFT coefficients of the voltage (i.e., if the 1^{st},3^{rd},5^{th}, and 7^{th} DFT coefficients of current are stored, then these same four coefficients of voltage should also be stored) By definition, time averaged real

power (over one line-cycle) is given by

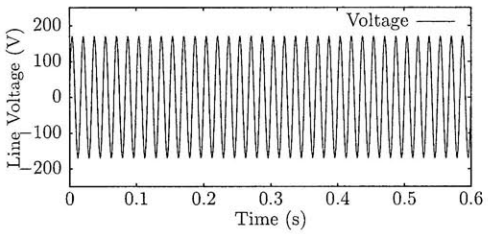
$$P = \frac{1}{N} \sum_{n=0}^{N-1} i[n]v[n],$$

where $i[n]$ and $v[n]$ are the samples of current and voltage, respectively, over one period. Let I_k and V_k denote the (complex) amplitudes of the k^{th} harmonics of current and voltage, respectively. Using the Plancherel Theorem, real power can be expressed in terms of the DFT of current and voltage,

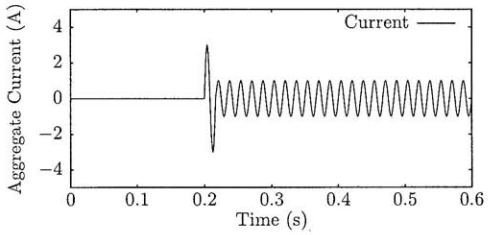
$$P = \frac{1}{N^2} \sum_{k=0}^{N-1} I_k V_k^*.$$

Reactive power can be calculated in an analogous fashion. Thus, if it is indeed true that only a small number of DFT coefficients of current are not approximately zero, then an accurate approximation of real and reactive power could be obtained by storing only the few significant DFT coefficients of current, and the same set of DFT coefficients of voltage.

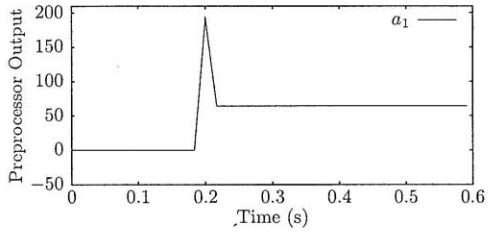
Figure 5.2, shown below, shows the line voltage, aggregate current, and preprocessor output, during the turn on of a device that draws exclusively real power. The only non-zero preprocessor envelope is the envelope that corresponds to real power. It is important to note that, because the only non-zero DFT coefficient of current is the 1^{st} coefficient (fundamental), this envelope is truly a scaled version of real power, regardless of the harmonic content of the voltage waveform. This is due to the fact that, by the above logic, the only harmonics of the voltage waveform that affect real and reactive power are those harmonics that are also present in current. For simplicity, this example, as well as all other example transients in the remainder of this section, consist of simulated data. However, it should be noted that, even though the analysis in this section is



(a) Line Voltage.



(b) Aggregate Current.



(c) Preprocessor Output.

Figure 5.2: Sample Preprocessor Output. The top two plots depict the measured line voltage and aggregate current, respectively, while a simulated device is being turned on. The third plot depicts the preprocessor output.

illustrated with simulated examples, the analysis applies equally well to real data.

Interesting examples arise from loads that draw non-sinusoidal or harmonically distorted current waveforms, like some personal computers or compact fluorescent lamps. Figure 5.3 depicts the current drawn from a simulated load over one period of the line voltage. This device consumes first and third harmonic current. Calculated preprocessor output for this simulated device is shown in Figure 5.4.

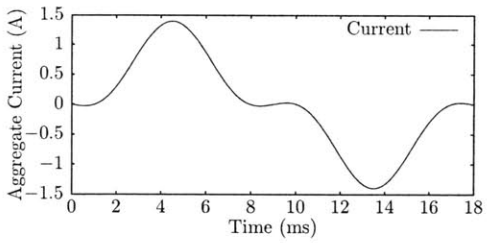
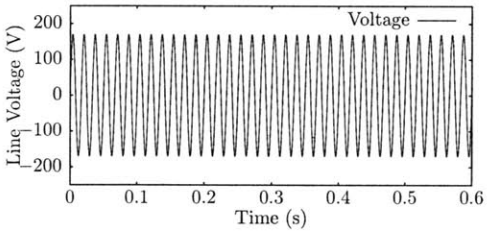
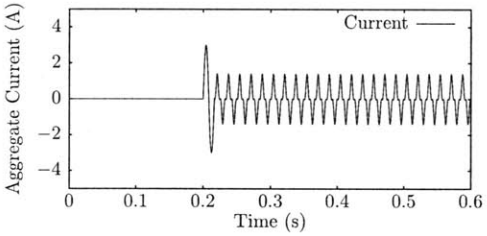


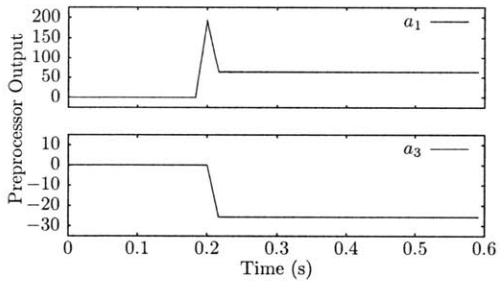
Figure 5.3: View of current over one period of the line voltage. This simulated device draws current predominantly at the first and third harmonic.



(a) Line Voltage.



(b) Aggregate Current.



(c) Preprocessor Output.

Figure 5.4: Sample Preprocessor Output. The first two plots show the measured line voltage and aggregate current, respectively. The third pair shows the preprocessor outputs a_1 and a_3 corresponding to in-phase current draw at the first and third harmonics.

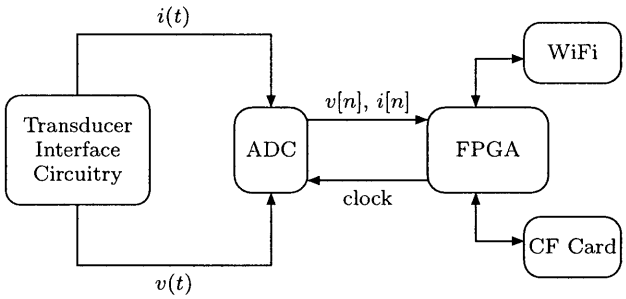


Figure 5.5: Spectral Envelope Preprocessor Block Diagram. An Analog-to-Digital converter is used to produce samples of the line voltage and the aggregate current, which are then used by the FPGA to produce spectral envelope coefficients. These coefficients can be stored in a compact flash card and also transmitted via 802.11 WiFi on demand.

5.3 FPGA-Based Spectral Envelope Preprocessor

To calculate, store, and communicate a relevant subset of DFT coefficients for power monitoring and energy scorekeeping, a prototype FPGA (Field Programmable Gate Array) was constructed to implement a spectral envelope preprocessor. All Verilog code can be found in Appendix C. This system makes use of a low-cost FPGA (Altera Cyclone I, EP1C3T100C8). The spectral preprocessor consists of four subsystems: a subsystem that obtains current and voltage samples, a subsystem that computes spectral envelope coefficients, a subsystem that stores computed spectral envelope coefficients, and a subsystem that can transmit the spectral envelope coefficients to another computation or display platform for further analysis. Each of these subsystems will be considered in detail. Figure 5.5 shows the overall block diagram of the system.

Data flows through the system as follows. The transducer interface circuitry measures the line voltage and aggregate current, producing the signals $v(t)$ and $i(t)$. These signals are sampled and quantized by an analog-to-digital converter (ADC) that produces the samples $v[n]$ and $i[n]$. The FPGA processes these samples to compute spectral envelopes. The spectral envelope coefficients can be stored in a Compact Flash (CF)

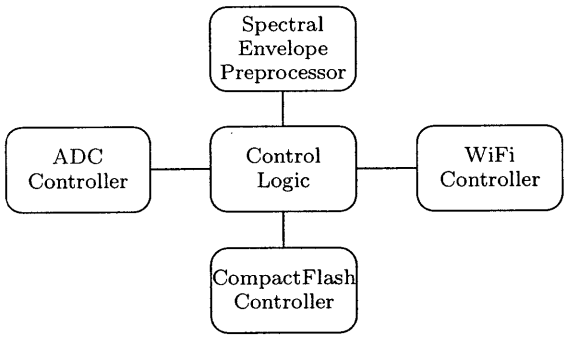


Figure 5.6: FPGA Block Diagram. The preprocessor is used to calculate spectral envelope coefficients. The ADC controller is used to control the sampling scheme of the Analog-to-Digital converter. The CF controller interfaces with a Compact Flash card to enable spectral envelopes to be stored and later recalled. The WiFi controller interfaces with an 802.11 WiFi transceiver to transmit spectral envelope data.

card for later use. The system also includes an 802.11b/g WiFi transceiver that allows any collection of the spectral envelope coefficients to be transmitted to another computation device for analysis. The FPGA provides control logic for each of the subsystems. Figure 5.6 shows a block diagram of the system implemented in the FPGA.

5.3.1 Current and Voltage Measurement

Current and voltage measurements from at least one voltage channel and at least one current channel are used to compute spectral envelopes. The system is easily expanded to measure more than two channels, supporting three-phase electrical services, for example. The prototype system uses an LA-55 current transducer to measure aggregate current and a simple transformer to measure the line voltage. A transformer with dual secondary coils was used in the prototype. This provides one coil for measurement purposes, and a second coil for powering the preprocessor. The two coil arrangement provides a voltage sense with very little phase distortion, ensuring accurate calculation of in phase and quadrature spectral components. Figure 5.7 illustrates this utility connection.

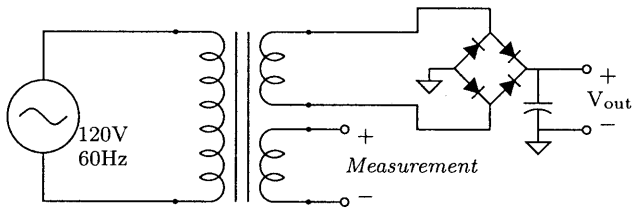


Figure 5.7: The FPGA system receives both line voltage measurement and low-voltage supply through a transformer with dual secondary coils.

5.3.2 ADC Controller

In many signal processing applications, a computationally efficient algorithm like the Fast Fourier Transform (FFT) computes the complete spectral analysis of a sampled waveform. However, in situations like power monitoring, where a relatively small number of spectral coefficients may contain all or most needed information, needed spectral coefficients can be computed more efficiently by a traditional DFT implementation, i.e., by mixing observed waveform samples directly with the stored samples of basis sinusoids. In this approach, basis sinusoids are stored in a memory and multiplied by observed samples of a waveform. If there are N samples stored in memory for each basis sinusoid, then it is necessary to acquire N samples of the current and voltage waveforms for each line voltage period.

The FPGA coordinates the operation of the ADC (Analog to Digital Converter) to obtain the samples $i[n]$ and $v[n]$ of the current and voltage waveforms $i(t)$ and $v(t)$. To provide a known number of waveform samples per line period, the FPGA "locks" to the line voltage waveform. That is, the FPGA varies the sample rate to track with variations in the line voltage frequency. The sampling clock is derived from the output of a digital phase-locked loop (PLL) on the FPGA that tracks the line voltage frequency.

The phase of the sampling is set such that the first voltage and first current samples are taken at the negative to positive zero crossing of the line voltage. The goal is

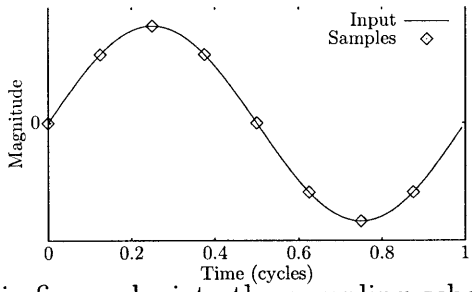


Figure 5.8: Sampling. This figure depicts the sampling scheme used to obtain samples of the line voltage and aggregate current. Samples are taken at points in time that correspond to the samples stored in the basis sinusoid memory. For clarity, this figure only depicts 8 sample points per cycle, while the prototype system actually uses 128 sample points per cycle.

to multiply each entry in the basis sinusoid by the value of the waveform to be analyzed at the corresponding point in time. It is essential that the entire process is line locked to the line frequency in order for the estimated spectral envelope coefficients of current to correspond to "in-phase" and "quadrature" components of current with respect to the fundamental of the voltage waveform. This sampling scheme is illustrated in Figure 5.8. The sample times and values are indicated by diamonds.

5.3.3 Envelope Preprocessor

Many interesting hardware and software systems that can calculate spectral envelope or quantities related to spectral envelopes have previously been constructed. References [14] – [16] and their associated references describe various metering schemes that compute real, reactive, and apparent power, and also harmonic distortion in one form or another in many cases. In [1], multiple phase-locked loops, analog multipliers and integrators were used to estimate spectral envelope coefficients. In [10], a design using multiplying digital-to-analog converters, low-pass filters, and a single phase-locked loop was presented. In [11], a expensive digital signal processing board was used to perform the calculations. In [12], the processing power of a personal computer was used for

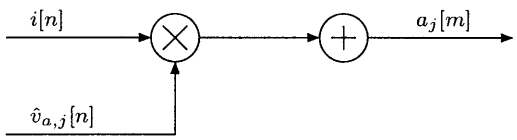


Figure 5.9: Signal Flow Graph. This diagram depicts the signal path for the calculation of a spectral envelope. Raw current values are multiplied by the appropriate elements of each basis sinusoid, and the results are accumulated over each period to produce each spectral envelope value.

spectral envelope coefficient estimation.

All of these systems can provide accurate estimates of spectral envelope coefficients or related quantities. They serve as essential building blocks of various types of metering systems. They are often expensive and dedicated. The FPGA-based system discussed in this section is an inexpensive single-chip solution that can estimate spectral envelope coefficients for stand-alone use or as part of a turn-key building block in more complex systems. The FPGA computes spectral envelopes using integer arithmetic on stored basis waveforms and observed waveform samples.

The FPGA-based spectral envelope preprocessor calculates the spectral envelopes of current, $a_1[m], b_1[m], \ldots$, where m indexes the periods of the line voltage. Figure 5.9 shows the computation performed to produce estimates of a single spectral envelope coefficient, $a_j[m]$. The system multiples $i[n]$, the samples of current, with $\hat{v}_{a,j}[n]$, the samples of a basis sinusoid, and sums the result over a single period of the line voltage. If N denotes the number of samples per period, then

$$a_j[m] = \sum_{k=0}^{N-1} i[mN+k] \cdot \hat{v}_{a,j}[mN+k] \tag{5.3}$$

and similarly,

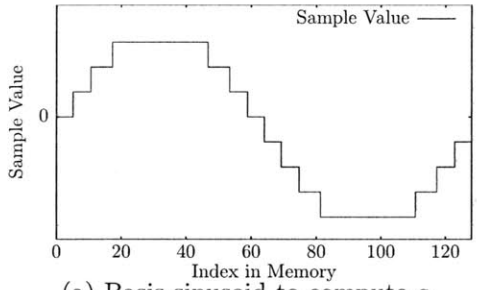
$$b_j[m] = \sum_{k=0}^{N-1} i[mN+k] \cdot \hat{v}_{b,j}[mN+k] \tag{5.4}$$

Each spectral envelope coefficient has a different basis sinusoid associated with it; for example, calculation of $a_3[m]$ involves multiplying $i[n]$ by $\hat{v}_{a,3}[n]$, where $\hat{v}_{a,3}[n]$ consists of discrete time samples of a sinusoid at three times the line frequency, with its phase locked to the line voltage. Figure 5.10, shown below, depicts examples of basis sinusoids. For illustration purposes, these sinusoids are sampled at only 3 bits, while the prototype system makes use of 10 bit samples.

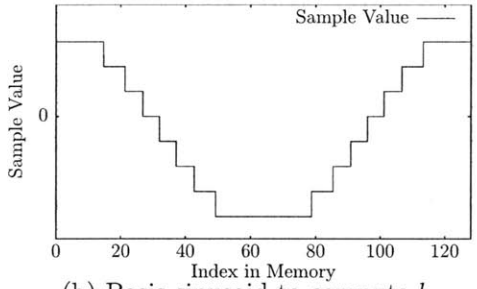
Figure 5.11 shows a block diagram of the FPGA-based spectral envelope preprocessor. The preprocessor takes the discrete time samples of $i(t)$ and $v(t)$ as input, denoted $i[n]$ and $v[n]$ respectively, where n indexes the samples, and produces estimates of the spectral envelope coefficients a_j and b_j of the current $i[n]$.

The voltage samples $v[n]$ are used as input to a phase-locked loop (PLL), which synchronizes the entire computation to the line voltage. As noted earlier, the computation process is synchronized to the line voltage so that the calculated spectral envelope coefficients correspond to some extent to meaningful physical quantities (real power, reactive power, etc.). The output of the PLL is sent to a block of steering logic on the FPGA that produces the address for the basis sinusoid memory, as well as a clear signal for the accumulators. The basis sinusoid memory consists of the samples of the various basis sinusoids. The address produced by the steering logic specifies a single sample time of a single basis sinusoid. The sample of the basis sinusoid that is retrieved from the basis sinusoid memory is then multiplied by the current sample $i[n]$. The result of this multiplication is then passed through a demultiplexer which sends the result to the appropriate accumulator by using the address produced by steering logic to determine which spectral envelope coefficient is currently being calculated.

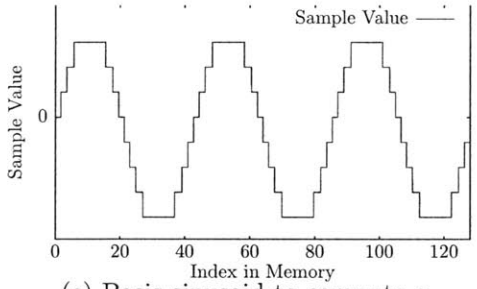
There is one accumulator for each estimated spectral envelope coefficient. The accumulators are all cleared at the end of each period of the line voltage through the



(a) Basis sinusoid to compute a_1



(b) Basis sinusoid to compute b_1



(c) Basis sinusoid to compute a_3

Figure 5.10: Basis Sinusoids. This figure depicts three examples of basis sinusoids, used to calculate real spectral envelopes of in-phase fundamental frequency content, quadrature fundamental frequency contents, and in-phase third harmonic content, from top to bottom, respectively. The basis sinusoids shown here are sampled at 3 bits for illustration – the actual prototype spectral envelope preprocessor uses 10 bit samples.

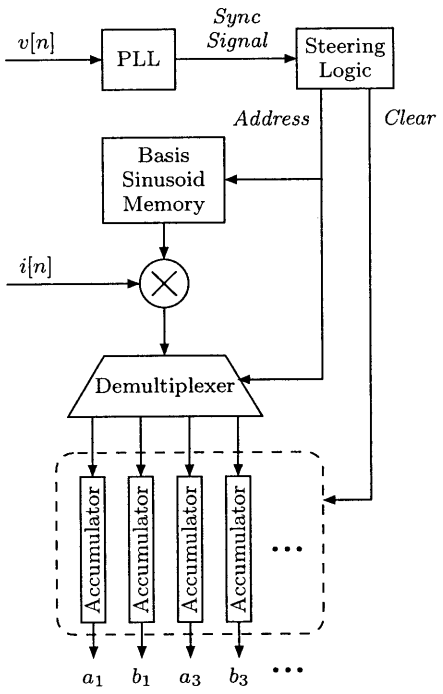


Figure 5.11: Preprocessor Block Diagram.

use of the clear signal produced by the steering logic. For every sample $i[n]$, the address produced by the steering logic will select each of the basis sinusoids in turn, so that every sample of current is multiplied by the appropriate sample of each of the basis sinusoids.

This FPGA-based implementation provides a great deal of flexibility. For example, the subset of spectral envelope coefficients that are being estimated can be changed by altering the entries in the memory to correspond to a different set of basis sinusoids. This implementation is also efficient in terms of FPGA resource utilization. It uses only a single PLL and a single multiplier, as opposed to previous hardware implementations that often used multiple PLLs and/or multipliers [1],[10]. This system can function with only a single multiplier because the FPGA is capable of multiplying each sample $i[n]$ by the corresponding sample of each of the basis sinusoids and forwarding each result to the appropriate accumulator, before the next sample $i[n+1]$ arrives. The multiplier consumes substantial logic elements on the FPGA. It consumes 24% of all resources used

by the envelope preprocessor and 13% of all resources used by the complete system. By using only one multiplier, the design is capable of fitting in a small, low-cost FPGA.

There are several ways to configure and deploy the spectral envelope preprocessor for any given application. For situations where the voltage waveform is relatively sinusoidal and "stiff," the spectral envelopes of current can be interpreted as scaled physical quantities in steady state. As discussed above, under these assumptions, the a_1 envelope of current in steady state corresponds to a scaled estimate of real power or "P". The b_1 envelope of current corresponds to reactive power or "Q". In situations where the voltage is not stiff and/or not sinusoidal, the FPGA could be tasked to also compute the spectral envelopes of voltage as well as current. This more complete set of spectral envelopes could be stored or transmitted to a computation platform or metering instrument that can quickly compute estimates of real or reactive power or other quantities of interest. Alternatively, the FPGA can be reconfigured to compute quantities like real and reactive power. In practice, it has been found that the basic computation of the spectral envelopes of current, assuming a stiff voltage source, to yield information that is directly useful for energy scorekeeping and demand-side load control and diagnostics, e.g., see [1].

5.3.4 CF Controller

The purpose of this FPGA subsystem is to store spectral envelope data on an erasable memory like a CF (Compact Flash) storage card. This subsystem is capable of storing spectral envelope data as it is produced, as well as retrieving the spectral envelope data from any point in time, on demand. To interface with the CF card, the "True IDE" interface mode is used. This interface mode is universally supported by compact flash storage cards and it allows the system to be easily adapted to interface with other mass

storage devices that use the IDE interface standard, such as an IDE hard drive. While it would be possible to impose a filesystem on the CF card (i.e. FAT32), the current design treats the CF card as a single large, raw block of storage, for simplicity. Due to the low data rate of the spectral envelope coefficients (for the prototype preprocessor with 8 spectral envelopes, each stored at 24 bits of resolution, the data rate is 2.8 KB/s), even a moderately sized CF card could store the spectral envelope data for a substantial length of time. For example, in for the prototype system, a 1 GB CF card would suffice to store data for approximately 4.3 days.

5.3.5 WiFi Controller

This subsystem facilitates the transmission of spectral envelope coefficients to a PC or other computation platform for further analysis or display. It makes use of an 802.11b/g WiFi transceiver and TCP/IP. The current design is capable of supporting both ad-hoc and access point (infrastructure) networks.

The transmitted data is retrieved from the CF card as needed. The WiFi subsystem can operate in two different modes. In the first mode, the system streams spectral envelope coefficients as they are generated. In the prototype, this corresponds to a data rate of 2.8 KB/s. In the event of a momentary interruption in the connection to the PC, the system will automatically buffer data from the last successfully transmitted packet and resume transmission from that point when a connection is reestablished. The system will then send data at the highest available transmit speed (54 Mb/s for 802.11g), until the system catches up to the freshly produced spectral envelope data. In the second mode, an application on the PC requests data by specifying a range of time; the system then transmits all data from the desired range of time, at the maximum possible

transmission speed.

5.4 Flexibility

The design presented above is just one example of an FPGA-based load monitoring interface. The modularity of the design, and the versatility of FPGAs, makes it simple to change the transmission system, for example, to wired ethernet (IEEE 802.3) or Bluetooth (IEEE 802.15.1) or ZigBee, or to change the storage system to, for example, a microSD card. An FPGA permits the interconnection of a wide variety of different subsystems to form a complex utility monitoring system. Even a small, low-cost FPGA is capable of implementing both the spectral envelope preprocessor as well as the required interface logic to control the various subsystems. Thus, an FPGA can serve as the backbone of an inexpensive, complete utility or load monitoring system.

5.5 Prototype Results

The FPGA-based system discussed above calculates, stores, and transmits spectral envelope data. Figure 5.12, shown below, shows a picture of the prototype hardware.

To make use of this data, a monitoring or control system typically includes a subsystem to receive and use the spectral envelope data. For example, a homeowner could use a personal computer to collect spectral envelope data from the FPGA preprocessor installed near or in a circuit breaker panel. The prototype includes a PC-based software application that can interface with the FPGA-based preprocessor via wired or wireless communication channels, retrieve spectral envelopes, and display spectral envelope data. Using this spectral envelope data, the PC-side application can disaggregate the operating

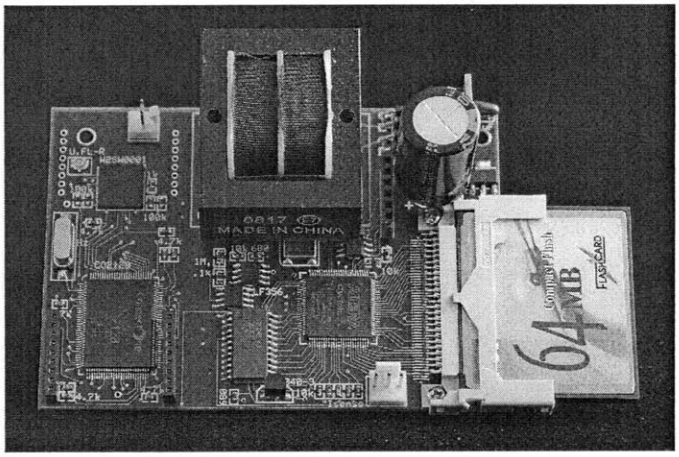


Figure 5.12: Prototype hardware. This is a picture of the prototype FPGA-based system.

schedule of individual loads from measurements made on an aggregate power feed serving multiple loads. The application is self-training and identifies loads in essentially real-time. Screenshots of this program are shown in Figure 5.13 and Figure 5.14.

This software communicates via TCP/IP with the FPGA-based preprocessor. The software can retrieve any subset of recorded data, as well as issue commands, such as changing the sampling resolution of the ADC. Once spectral envelope data is retrieved, this software makes use of the Expectation-Maximization (EM) algorithm [8] to classify or recognize the operation of individual loads.

This software is only one example of the many possible ways to use spectral envelope data. Other software applications that make use of spectral envelope data could be developed (a system that uses this data to control a set of generators in a micro-grid is currently in development). Data could be retrieved by a web application that displays a live stream of data on a webpage. Other embedded systems could communicate with the FPGA-based system and use the retrieved spectral envelope data to control electrical loads.

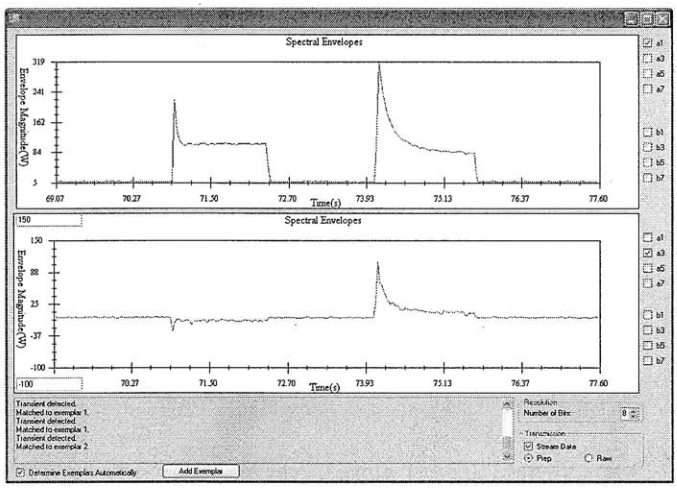


Figure 5.13: Screenshot. This figure shows a screenshot of the prototype non-intrusive monitoring software in operation. There are two plots in the figure that display spectral envelope data. The upper plot displays real power and the lower plot displays third harmonic content. The spectral envelope data corresponds to the light bulb and motor whose raw current values are shown in Figure 5.1. The lower left section of the screenshot shows the output of the classifier, which has correctly identified both of the loads.

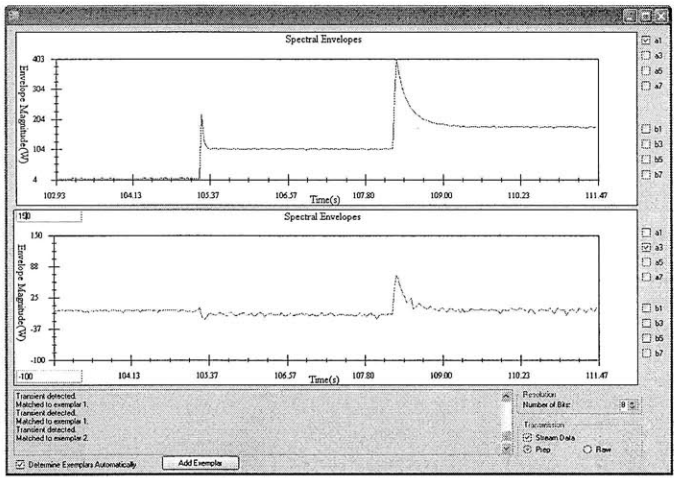


Figure 5.14: Screenshot. This figure shows a screenshot of the prototype non-intrusive monitoring software in operation. As in Figure 5.13, the upper plot displays real power and the lower plot displays third harmonic content. The data corresponds to the same light bulb and motor as Figure 5.13, with the difference being that now the motor is turned on while the light bulb is on. As shown in the lower left corner of the screen, the system still classifies both devices correctly.

5.6 Applications

The FPGA preprocessor can provide a turn-key component for creating all sorts of utility monitoring, energy score-keeping, and diagnostic applications for all sorts of systems. The preprocessor is relatively simple compared to microprocessor or DSP-based data acquisition systems. The concepts and hardware illustrated here could be incorporated into individual loads, circuit breakers, or circuit breaker panels to provide energy consumption information for both monitoring and control.

The simplification in data storage and transmission bandwidth requirements afforded by the FPGA can be extended to other domains and monitoring problems. For example, it is possible to extend the non-intrusive monitoring concept beyond the realm of electrical distribution. A single acoustic sensor could be used to monitor the flow of water, for example, in a main water service to a set of rooms in a building. Fingerprint acoustic signatures can be developed that permit recognition of hydraulic loads or events in the water distribution system. This acoustic data is not "line locked" to any particular "utility frequency." However, much like a voice signal, acoustic data can be described by simplifying expressions, e.g., the coefficients of a time series, also known as linear predictor coefficients (LPCs) [18]. The FPGA could be tasked to compute these LPCs and transmit them, again providing a significant bandwidth reduction for storage or transmission. Other applications may also be possible.

The approach demonstrated in this chapter permits a flexible trade-off between the hardware installed proximal to a monitored device or collection of devices and the transmission bandwidth to and remote computation capability at a distal monitoring or information gathering system. An FPGA like the one described here could serve as a central coordinator for gathering, processing, and transmitting all sorts of utility

information, including simultaneous monitoring of electrical, water, and gas services. This type of monitoring can support home or building level energy conservation and diagnostics efforts. It might also be useful for coordinating the operation of generation and the scheduling of demand on micro-grid power distribution systems or similar power distribution systems on transportation systems.

Chapter 6

Conclusion

The techniques illustrated in this thesis can be applied to a wide range of signal processing problems, including non-intrusive load monitoring. The FPGA-based spectral envelope preprocessor, discussed in Chapter 5, provides an inexpensive, accurate, and convenient platform for collected a variety of useful data about a collection of electrical loads. This information can be used for a variety of power monitoring and energy scorekeeping tasks, as well as to diagnose problems with individual electrical loads. The algorithms presented in Chapters 2-4 can be applied to enhance the capabilities of a NILM system.

Chapter 4 presented an algorithm that could identity a single electrical load from a collection of electrical loads, by examining a subset of the spectral envelopes of the current drawn by the unknown load. This allows the data collected by the FPGA-based system of Chapter 5 to be used to non-intrusively monitor a collection of electrical loads and determine when each load is turned on and off, as well as how much power each load consumed at any point in time.

Chapter 3 considered the problem of using knowledge of one subset of spectral envelope values to estimate another subset of spectral envelope values, for an appropriately constrained class of waveforms. A simple but numerically unstable algorithm to solve this problem was first presented, followed by a refined approach that avoided numerical instability to exploiting properties of cyclotomic fields. In a NILM environment, many simultaneously operating loads may draw currents that have partially overlapping harmonic content. The algorithm presented in this section would allow the estimation of all spectral envelopes of each individual load by using the band of harmonic content that is unique to that load to estimate the overlapping portions of harmonic content.

Chapter 2 examined the problem of calculating the DFT of a quantized signal. An algorithm was presented that used the structure of the mapping between regions of frequency space and quantized current to accurately, and efficiently, estimate the true spectral envelope values of a measured current. This algorithm is invaluable when dealing with data produced by the FPGA-based NILM of chapter 5 as it allows accurate estimates of true power consumption to be made from the quantized data collected by that system.

The algorithms of Chapters 2-4 can be applied to a variety of discrete-time signal processing tasks that involve the computation of the DFT of a signal. The algorithm of chapter 2 can be applied to any situation in which one desires accurate computation of the DFT of a signal, but is only provided with a coarsely quantized version of that signal. The algorithms in chapter 3 can be applied to a variety of estimation problems where the constraints can be written with coefficients in a cyclotomic field. Finally, the classification algorithms presented in chapter 4 could be applied in other classification contexts when the objects being classified have distinct harmonic signatures.

Appendix A

Matlab Code for DFT Accuracy Improvement

This appendix consists of Matlab code that implements the algorithm discussed in Chapter 2. There are 3 functions: findFirstVertex which finds a single vertex of a region R, findNeighbors which finds all neighbors of vertex x of region R and findAllVertices which finds all vertices of region R.

```
function [x]=findFirstVertex(y,A,B)
    err=10^(-4);
    x=y;
    [numConst,numHarms]=size(A);
    S=zeros(0,numHarms);
    D=zeros(0,1);
    C=zeros(0,1);
    s=rand(1,numHarms)-.5;
    for i=1:numHarms
```

```
        dists=zeros(1,numConst);
        for j=1:numConst
            if(isempty(find(C==j,1)))
                init=sum(A(j,:).*x);
                adj=sum(A(j,:).*s);
                dists(j)=(B(j)-init)/adj;
                if(dists(j)<=err)
                    dists(j)=inf;
                end
            else
                dists(j)=inf;
            end
        end
        [dummy,newConstraint]=min(dists);
        x=x+dists(newConstraint)*s;
        if(i<numHarms)
            C=vertcat(C,newConstraint);
            S=vertcat(S,A(newConstraint,:));
            D=vertcat(D,[B(newConstraint)]);
            [currRows,dummy]=size(S);
            s=(vertcat(S,rand(numHarms-currRows,numHarms))\
                vertcat(D,zeros(numHarms-currRows,1)))'-x;
        end
    end
```

```
function [L]=findNeighbors(y,A,B)
    err=10^(-6);
    [numCons,numHarms]=size(A);
    L=zeros(0,numHarms);
    satCons=L;
    unsatCons=L;
    satD=zeros(0,1);
    unsatD=zeros(0,1);
    isUpper=zeros(0,1);
    for i=1:numCons
        if(abs(sum(A(i,:).*y)-B(i))<err)
            satCons=vertcat(satCons,A(i,:));
            satD=vertcat(satD,B(i));
            isUpper=vertcat(isUpper,(i<=numCons/2));
        else
            unsatCons=vertcat(unsatCons,A(i,:));
            unsatD=vertcat(unsatD,B(i));
        end
    end
    dists=zeros(1,length(unsatD));
    for i=1:length(satD)
        if(isinf(satD(i))==0)
            for j=i+1:length(satD)
                if(isinf(satD(j))==0 && ( (sum(abs(satCons(i,:)-
                    satCons(j,:)))<err && abs(satD(i)-satD(j))<err
```

```
                ) || (sum(abs(satCons(i,:)+satCons(j,:)))<err
                && abs(satD(i)+satD(j))<err)))
                  satD(j)=inf;
            end
        end
    end
end
goodRows=find(1-isinf(satD));
satCons=satCons(goodRows,:);
satD=satD(goodRows);
isUpper=isUpper(goodRows);
for i=1:length(satD)
    tempCons=satCons;
    tempCons(i,:)=rand(1,numHarms);
    tempD=satD;
    tempD(i)=0;
    dir=(tempCons\tempD)'-y;
    if((sum(dir.*satCons(i,:)))*(2*isUpper(i)-1)>0)
        dir=-dir;
    end
    for j=1:length(unsatD)
        init=sum(unsatCons(j,:).*y);
        adj=sum(unsatCons(j,:).*dir);
        dists(j)=(unsatD(j)-init)/adj;
        if(dists(j)<err)
```

```
            dists(j)=inf;
        end
    end
    [newDist,newConstraint]=min(dists);
    candL=y+newDist*dir;
    candBad=0;
    sL=size(L);
    for j=1:sL(1)
        if(sum(abs(L(j,:)-candL))<err)
            candBad=1;
        end
    end
    if(candBad==0)
        L=vertcat(L,y+dists(newConstraint)*dir);
    end
end
```

```
function [V]=findAllVertices(y,A,B)
    err=10^(-6);
    [numCons,numHarms]=size(A);
    V=zeros(0,numHarms);
    H=[findFirstVertex(y,A,B)];
    sH=size(H);
    while(sH(1)>0)
        V=vertcat(V,H);
        L=zeros(0,numHarms);
        for i=1:sH(1)
            newL=findNeighbors(H(i,:),A,B);
            sNL=size(newL);
            sV=size(V);
            for j=1:sNL(1)
                if(sV(1)==0 || min(sum(abs(V-repmat(newL(j,:),sV
                    (1),1)),2))>err)
                    L=vertcat(L,newL(j,:));
                    sL=size(L);
                end
            end
        end
        H=L;
        sH=size(H);
    end
```

Appendix B

GP/PARI Code for cross estimation

The code in this appendix was developed in collaboration with Warit Wichakool. This appendix consists of GP/PARI code that implements the algorithms discussed in chapter 3. There are 5 functions: nfrref.gp computes the RREF of a matrix with entries that are elements of a number field, ntt.gp computes the number theoretic transform, fntt.gp computes a "fast" number theoretic transform, mul.gp multiplies polynomials using the NTT and div.gp divides polynomials using the NTT.

```
nfrref(A, OptionRowEchelonForm)=
{
local(numRows, numCols, colIndex, rowIndex, rowCounter, found, M, temp,
    pivotVal, k, lNeigh);

M=A;
numRows=length(A[,1]);
numCols=length(A);
colIndex=1;
```

```
rowCounter=1;

while((colIndex<=numCols) && (rowCounter<=numRows),

    rowIndex=rowCounter;
    found=0;

    while(rowIndex<=numRows && !found,
        if(M[rowIndex,colIndex]==0,rowIndex++;,found=1;);
    );

    if(found,
        if(rowCounter!=rowIndex,
                temp=M[rowIndex,];
                M[rowIndex,]=M[rowCounter,];
                M[rowCounter,]=temp;
        );

        pivotVal=M[rowCounter,colIndex];
        M[rowCounter,] = M[rowCounter,]/pivotVal;

        for(k=1,numRows,
                if((OptionRowEchelonForm && ((k == 1 &&
                    rowCounter != 1) || k > rowCounter)) || (!
                    OptionRowEchelonForm && k!=rowCounter),
```

```
                        lNeigh=M[k, colIndex];
                        M[k,]=M[k,]-M[rowCounter,]*lNeigh;
                    );

            );
            rowCounter++;
        );
        colIndex++;
);
return(M);
}
```

```
ntt(n,p,z,v)=
{

local(index,k,vmod,zmod,vout);

vout = vector(n);
vmod = Mod(v,p);
zmod = Mod(z,p);

for(index = 1,length(vout),
        for(k = 0, n-1,
                if(v[k+1],vout[index] += vmod[k+1]*zmod^lift(Mod
                    ((index-1)*k,(p-1))););
        );
);

return(lift(vout));
}
```

```
fntt(n,p,z,v,split)=
{

local(n1, n2, m, mhat, bx, row, col, z1, z2, modz1, modz2, k1,
   k2, k, modz, vout, vtemp, THRESHOLD);
THRESHOLD = 32;

if(split[n] == 0 || n < THRESHOLD,
        return(ntt(n,p,z,v));,

        n1 = split[n][1];
        n2 = split[n][2];
        modz=Mod(z,p);
        modz1 = Mod(z^(n2),p);
        modz2 = Mod(z^(n1),p);
        z1 = lift(modz1);
        z2 = lift(modz2);
        m = matrix(n2,n1,row,col,v[(row-1)*n1+(col-1)+1]);
        mhat = matrix(n1,n2);

        for(k = 1,n1,
            mhat[k,] = lift(Mod(fntt(n2,p,z2,m[,k],split),p)
                );
        );
```

```
        vout = vector(n);
        vtemp = matrix(n1,n2);

        for(k1 = 0, n1-1,
                for(k2 = 0, n2-1,
                        mhat[k1+1,k2+1] =lift(mhat[k1+1,k2+1]*
                            modz^(k1*k2));
                );

        );

        for(k2 = 0, n2-1,
                vtemp[,k2+1] = lift(Mod(fntt(n1,p,z1,mhat[,k2
                    +1],split),p))~;
        ); /* end for(k2 = 0, ...) */

        for(k1 = 0, n1-1,
                for(k2 = 0, n2-1,
                        vout[n2*k1 + k2 + 1] = vtemp[k1+1,k2+1];
                );
        );

        return(vout);
);
```

```
}
```

```
mul(a,b,polyTemp,n,ninv,p,z,zinv,sp)=
{
local(acoeff,bcoeff,ahat,bhat,chat,index);

acoeff = Vec(lift(a));
bcoeff = Vec(lift(b));

ahat = fntt(n,p,z,concat(vector(n-length(acoeff)),acoeff),sp);
bhat = fntt(n,p,z,concat(vector(n-length(bcoeff)),bcoeff),sp);
chat = vector(n);

for(index=1,n,chat[index]=lift(Mod(ahat[index]*bhat[index],p)));

d = fntt(n,p,zinv,chat,sp);
c = lift(Mod(ninv*d,p));
c = vecextract(c,"1..-2");
c = lift(Mod(Vec(lift(Mod(Pol(c),polyTemp))),p));
c = apply(x->if(x > (p-1)/2, x-p,x),c);

return(Mod(Pol(c),polyTemp));
}
```

```
div(a,b,polyTemp,n,ninv,p,z,zinv,sp)=
{
local(acoeff,bcoeff,ahat,bhat,chat,index, c, d);

acoeff = Vec(lift(a));
bcoeff = Vec(lift(b));

ahat = fntt(n,p,z,concat(vector(n-length(acoeff)),acoeff),sp);
bhat = fntt(n,p,z,concat(vector(n-length(bcoeff)),bcoeff),sp);
chat = vector(n);

for(index=1,n,chat[index]=lift(Mod(ahat[index]/bhat[index],p)));

d = fntt(n,p,zinv,chat,sp);
c = lift(Mod(ninv*d,p));
c = concat(c,[0]);
c = lift(Mod(Vec(lift(Mod(Pol(c),polyTemp))),p));
c = apply(x->if(x > (p-1)/2, x-p,x),c);

return(Mod(Pol(c),polyTemp));
}
```

Appendix C

Verilog Code for FPGA-Based Spectral Envelope Preprocessor

This chapter includes all Verilog code used to implement the FPGA-Based Spectral Envelope Preprocessor discussed in Chapter 5.

```
////////////////////////////////////////////////////////////////////////////////

//Top level module of PowerMon                                    //
//                                                                //
//Version 1.6

   //
//Author: Zack Remscrim
                                                                  //
////////////////////////////////////////////////////////////////////////////////
```

```
module PowerMon(
  //clock
  OSC_25,
  //led
  OLED,
  //compact flash
  CF_CS0, CF_CS1, CF_A, CF_D,CF_IO_R,CF_IO_W,CF_REG,CF_RESET,
     CF_INTRQ,
  //ADC
  ADC_DB, ADC_CS, ADC_RD, ADC_WR, ADC_INTRQ, ADC_CLK,
  //Ethernet
  ETHER_RXD,ETHER_TXD,ETHER_RESET
  );

  //clock
  input OSC_25;          //25 MHz clock

  //led
  output OLED;        //output led, active high

  //compact flash
  output CF_CS0;       //compact flash CS0, active low
  output CF_CS1;       //compact flash CS1, active low
  output [2:0] CF_A;     //compact flash address
  inout [15:0] CF_D;     //compact flash databus
```

```
output CF_IO_R;      //compact flash read, active low
output CF_IO_W;      //compact flash write, active low
output CF_REG;       //not used, always tie to VCC
output CF_RESET;     //reset
input CF_INTRQ;      //interrupt request

//ADC
inout [7:0] ADC_DB;   //ADC databus
output ADC_CS;       //ADC chip select, active low
output ADC_RD;       //ADC read, active low
output ADC_WR;       //ADC write, active low
input ADC_INTRQ;     //ADC interrupt request, active low
output ADC_CLK;      //ADC clock

//Ethernet
input ETHER_RXD;     //serial input line from ethernet module
output ETHER_TXD;    //serial output line to ethernet module
output ETHER_RESET;   //ethernet reset, active low

//Begin global wires and regs
wire OLED;
wire adcEnable;   //enables adc, active high
wire cfEnable;    //enable cf controller, active high
wire sysClk;     //system clock
wire[7:0] microData;
```

```
wire microWrite;
wire debug;
//End global wires and regs

//Begin module instantiations

//resetGen module and connections
wire globalReset;
resetGen aResetGen(sysClk, globalReset);

//adc buffer
wire[15:0] adcBufferDataIn, adcBufferDataOut;
wire adcBufferWr, adcBufferRd, adcBufferInClk, adcBufferOutClk,
    adcBufferEmpty, adcBufferFull;
adcBuffer aAdcBuffer(adcBufferDataIn, adcBufferWr, adcBufferRd,
    adcBufferInClk, adcBufferOutClk, adcBufferDataOut,
    adcBufferEmpty, adcBufferFull);

//adc controller
wire[7:0] ADC_DB;
wire ADC_CS, ADC_RD, ADC_WR, ADC_INTRQ, ADC_CLK;
adcController aAdcController(OSC_25, globalReset, adcEnable,
    ADC_DB, ADC_CS, ADC_RD, ADC_WR, ADC_INTRQ, ADC_CLK,
    adcBufferDataIn, adcBufferWr, adcBufferInClk);
```

```
//ethernet buffer
wire[7:0] etherBufferIn,etherBufferOut;
wire etherBufferWr,etherBufferRd,etherBufferFull,
   etherBufferEmpty;
wire[9:0] etherUsedWords;
etherFifo aEtherFifo(etherBufferIn,etherBufferWr,etherBufferRd
   ,~sysClk,etherBufferOut,etherBufferFull,etherBufferEmpty,
   etherUsedWords);

//ethernet controller
wire etherSendRaw;
etherController aEtherController(~sysClk,globalReset,ETHER_RXD
   ,ETHER_TXD,ETHER_RESET,etherBufferRd,etherBufferOut,
   etherBufferEmpty,etherUsedWords,etherSendRaw,debug);


//compact flash read buffer
wire [15:0] cfBufferDataIn, cfBufferDataOut;
wire [7:0] cfBufferAddrIn,cfBufferAddrOut;
wire cfBufferWr;
//cfBuffer aCfBuffer(cfBufferDataIn,cfBufferWr,cfBufferAddrIn,
   cfBufferAddrOut,~sysClk,cfBufferDataOut);

//compact flash write buffer
wire [15:0] cfWriteBufferDataIn, cfWriteBufferDataOut;
```

```
wire [7:0] cfWriteBufferAddrIn, cfWriteBufferAddrOut;
wire cfWriteBufferWr;
cfBuffer bCfBuffer(cfWriteBufferDataIn, cfWriteBufferWr,
    cfWriteBufferAddrIn, cfWriteBufferAddrOut, ~sysClk,
    cfWriteBufferDataOut);

//compact flash controller
wire CF_CS0, CF_CS1, CF_IO_R, CF_IO_W, CF_REG, CF_RESET, CF_INTRQ,
    cfBusy, cfLoadWrite, cfIsReadMode, tempDebug;
wire[2:0] CF_A;
wire[15:0] CF_D;
wire[3:2] cfDebug;
wire[7:0] cfLoadDataIn;
cfController aCfController(~sysClk, globalReset, cfEnable,
    microWrite, microData, CF_CS0, CF_CS1, CF_A, CF_D, CF_IO_R,
    CF_IO_W, CF_REG, CF_RESET, CF_INTRQ, cfDebug, cfBufferDataIn,
    cfBufferAddrIn, cfBufferWr, cfBusy, cfIsReadMode,
    cfWriteBufferAddrOut, cfWriteBufferDataOut, tempDebug);

//prep
wire[7:0] prepV, prepI;
wire prepNewSample, prepNewEnvelope, prepDebug;
wire[127:0] prepEnvelopes;
prep aPrep(~sysClk, globalReset, prepV, prepI, prepNewSample,
    etherSendRaw, prepEnvelopes, prepNewEnvelope, prepDebug);
```

```
  //main controller
  mainController aMainController(sysClk, globalReset,
     adcBufferDataOut, adcBufferRd, adcBufferEmpty, adcBufferFull,
     cfBufferAddrOut, cfBufferDataOut, cfWriteBufferDataIn,
     cfWriteBufferWr, cfWriteBufferAddrIn, microData, microWrite,
     cfIsReadMode, cfBusy, cfEnable, etherBufferIn, etherBufferWr,
     etherBufferFull, prepV, prepI, prepNewSample, prepEnvelopes,
     prepNewEnvelope, adcEnable);

  //End module instantiations

  //Begin global assignments
  //assign adcEnable=~globalReset;
  assign OLED=debug;
  assign sysClk=OSC_25;
  assign adcBufferOutClk=~sysClk;
  //assign ETHER_RESET=1'b1;

  //End global assignments
endmodule

//this module is responsible for managing the global reset
   signal
```

```
//it assures that reset is asserted for the first 64k clock
   cycles
module resetGen(clk,resetOut);
  input clk; //system clk

  output resetOut;  //global reset, active high

  parameter numCycles=64000;
  reg[15:0] cycleCount;
  wire resetOut;

  assign resetOut=(cycleCount<numCycles);
  always @(posedge clk) begin
    cycleCount<=(cycleCount<numCycles) ? cycleCount+1'b1 :
       cycleCount;
  end
endmodule

//this module is responsible for overall system control
//it takes data from adc fifo, moves it to cf buffer, and uses
   cfController to write to cf card
module mainController(clk,reset,adcBufferDataOut,adcBufferRd,
   adcBufferEmpty,adcBufferFull,cfReadBufferAddrOut,
   cfReadBufferDataOut,cfWriteBufferDataIn,cfWriteBufferWr,
   cfWriteBufferAddrIn,microData,microWrite,cfIsReadMode,cfBusy,
```

```
 cfEnable, etherBufferDataIn, etherBufferWr, etherBufferFull,
 prepV, prepI, prepNewSample, prepEnvelopes, prepNewEnvelope,
 adcEnable);
input clk;                  //system clk, 25MHz
input reset;                //global reset;
input[15:0] adcBufferDataOut;    //output data from adc buffer
input adcBufferEmpty;          //high when adc buffer is empty
input adcBufferFull;           //high when adc buffer is full
input[15:0] cfReadBufferDataOut;  //output data from cf read
   buffer
input cfBusy;               //high when cf card is busy
input etherBufferFull;         //high when ethernet buffer is
   full
input[127:0] prepEnvelopes;      //prep envelopes
input prepNewEnvelope;         //high for one cycle when a new
   set of prep envelopes is available

output adcBufferRd;            //read line for adc buffer, active
    high
output[7:0] cfReadBufferAddrOut;  //address for read buffer of
    cf card
output[15:0] cfWriteBufferDataIn; //input date to cf write
   buffer
output cfWriteBufferWr;        //write line for cf write buffer
   , active high
```

```
output[7:0] cfWriteBufferAddrIn;  //address for write buffer
   of cf card
output[7:0] microData;          //databus to other modules
output microWrite;             //write line to other modules,
   active low
output cfIsReadMode;           //1 to read from cf Card, 0 to
   write to cf card
output cfEnable;              //enable signal to cf controller,
   active high
output[7:0] etherBufferDataIn;    //input data to ethernet
   buffer
output etherBufferWr;          //write line for ethernet buffer,
   active high
output[7:0] prepV;             //next voltage sample for prep
output[7:0] prepI;             //next current sample for prep
output prepNewSample;          //asserted for one cycle to inform
    prep module that a new sample is ready, active high
output adcEnable;             //asserted to enable adc
//output led;
//reg led;

reg adcBufferRd, cfWriteBufferWr,microWrite,cfIsReadMode,
   cfEnable,etherBufferWr,prepNewSample,adcEnable;
reg[7:0] cfWriteBufferAddrIn,microData,cfReadBufferAddrOut,
   etherBufferDataIn,prepV,prepI;
```

```
reg[15:0] cfWriteBufferDataIn;
reg[29:0] initCount;          //used for initialization
reg[2:0] adcState;            //state for adc buffer read fsm
reg pendingCfWrite;           //high when there is a pending
   write to cf card
reg cfWriteDone;             //high when cf write completes
reg oldPrepNewEnvelope;        //prepNewEnvelope delayed by one
    cycle
reg[3:0] cfState;            //state for cf write fsm
reg[27:0] cfLBA;             //sector address of cfCard
reg[15:0] env0,env1,env2,env3,     //prep envelopes
  env4,env5,env6,env7;
reg[2:0] etherState;          //state for ether write fsm
reg pendingEnv;              //1 when there are envelopes waiting
    to be written to wifi controller
reg[2:0] envCount;            //used to count envelopes
reg initDone;              //used to control initialization

//fsm state enum, adc
parameter adcWaiting=3'b000;
parameter adcRead0=3'b001;
parameter adcRead1=3'b010;
parameter adcWrite0=3'b011;
parameter adcWrite1=3'b100;
parameter adcWrite2=3'b101;
```

```
parameter adcWrite3=3'b110;
parameter adcUpdate=3'b111;

//fsm state enum, cfWrite
parameter cfWriteWaiting=4'b0000;
parameter cfWriteB0a=4'b0001;
parameter cfWriteB0b=4'b0010;
parameter cfWriteB1a=4'b0011;
parameter cfWriteB1b=4'b0100;
parameter cfWriteB2a=4'b0101;
parameter cfWriteB2b=4'b0110;
parameter cfWriteB3a=4'b0111;
parameter cfWriteB3b=4'b1000;
parameter cfWriteB4a=4'b1001;
parameter cfWriteB4b=4'b1010;
parameter cfWriteComDone=4'b1011;
parameter cfWriteWaitBusy=4'b1100;
parameter cfWriteEnd0=4'b1101;
parameter cfWriteEnd1=4'b1110;

//fsm state enum, ether write
parameter etherWait=3'b000;
parameter etherGet=3'b001;
parameter etherWrite0=3'b010;
parameter etherWrite1=3'b011;
```

```
parameter etherWrite2=3'b100;
parameter etherWrite3=3'b101;
parameter etherDone=3'b110;

parameter initPer=450000000; //the device doesn't begin
   collecting data until initPer cycles after reset
parameter maxEnv=3'h7;

//transfer data from adc buffer to cf buffer and ethernet
   buffer
always @(posedge clk) begin
  initCount<=(initCount<initPer) ? initCount+1'b1 : initCount;
  adcEnable<=(initCount==initPer);
  if((reset==1'b1) || (initCount<initPer)) begin
    //initCount<=20'b0;
    adcState<=adcWaiting;
    //pendingCfWrite<=1'b0;
    adcBufferRd<=1'b0;
    //cfWriteBufferWr<=1'b0;
    //cfWriteBufferAddrIn<=8'b0;
    prepNewSample<=1'b0;
  end
  else begin
    /*if(initCount<initPer) begin //do initializations, setup
       cfWriteBuffer for test
```

```
    cfWriteBufferAddrIn<=(initCount[2:0]==3'b100) ?
        cfWriteBufferAddrIn+1'b1 : cfWriteBufferAddrIn;
    cfWriteBufferDataIn<={8'hf0,initCount[10:3]};
    cfWriteBufferWr<=(initCount[2:0]==3'b111);
  end
  else begin*/
    prepNewSample<=(adcState==adcWrite0);
    adcBufferRd<=(adcState==adcRead0);
    //cfWriteBufferWr<=(adcState==adcWrite0);
    case(adcState)
      adcWaiting:    adcState<=(adcBufferEmpty==1'b0 &&
          pendingCfWrite==1'b0) ? adcRead0 : adcWaiting;
      adcRead0:    adcState<=adcRead1;
      adcRead1:    begin
                     {prepI,prepV}<=adcBufferDataOut;
                     adcState<=adcWrite0;
                   end
      adcWrite0:     adcState<=adcWrite1;
      adcWrite1:     adcState<=adcWrite2;
      adcWrite2:     adcState<=adcWrite3;
      adcWrite3:     adcState<=adcUpdate;
      adcUpdate:     adcState<=adcWaiting;
    endcase
  //end
end
```

```
end

//transfer data from cf buffer to cf card
//byte0: low 8 bits of sector address
//byte1: next higher 8 bits of sector address
//byte2: next higher 8 bits of sector address
//byte3: 4 zeros, high 4 bits of sector address
//byte4: the literal 01h
always @(posedge clk) begin
  cfIsReadMode<=1'b0;. //temporary, for now always write
  //led<=(cfState==cfWriteWaitBusy);
  if(reset) begin
    cfWriteDone<=1'b1;
    cfLBA<=28'h0000101;
    cfEnable<=1'b0;
    microWrite<=1'b1;
    cfState<=cfWriteWaiting;
  end
  else begin
    cfEnable<=~(cfState==cfWriteEnd1);
    case(cfState)
      cfWriteWaiting: begin //wait for pending write to be
          asserted and cfBusy to clear
              cfState<=(pendingCfWrite==1'b1 && cfBusy==1'b0
                  ) ? cfWriteB0a : cfWriteWaiting;
```

```
            microWrite<=1'b1;
        end
cfWriteB0a:    begin //write b0
            cfWriteDone<=1'b0;
            cfState<=cfWriteB0b;
            microData<=cfLBA[7:0];
            microWrite<=1'b0;
        end
cfWriteB0b:    begin //write b0
            cfWriteDone<=1'b0;
            cfState<=cfWriteB1a;
            microData<=cfLBA[7:0];
            microWrite<=1'b1;
        end
cfWriteB1a:    begin //write b1
            cfState<=cfWriteB1b;
            cfWriteDone<=1'b0;
            microData<=cfLBA[15:8];
            microWrite<=1'b0;
        end
cfWriteB1b:    begin //write b1
            cfWriteDone<=1'b0;
            cfState<=cfWriteB2a;
            microData<=cfLBA[15:8];
            microWrite<=1'b1;
```

```
            end
cfWriteB2a:    begin //write b2
               cfState<=cfWriteB2b;
               cfWriteDone<=1'b0;
               microData<=cfLBA[23:16];
               microWrite<=1'b0;
            end
cfWriteB2b:    begin //write b2
               cfWriteDone<=1'b0;
               cfState<=cfWriteB3a;
               microData<=cfLBA[23:16];
               microWrite<=1'b1;
            end
cfWriteB3a:    begin //write b3
               cfState<=cfWriteB3b;
               cfWriteDone<=1'b0;
               microData<={4'h0,cfLBA[27:24]};
               microWrite<=1'b0;
            end
cfWriteB3b:    begin //write b3
               cfWriteDone<=1'b0;
               cfState<=cfWriteB4a;
               microData<={4'h0,cfLBA[27:24]};
               microWrite<=1'b1;
            end
```

```
cfWriteB4a:    begin //write b4
                cfState<=cfWriteB4b;
                cfWriteDone<=1'b0;
                microData<=8'h01;
                microWrite<=1'b0;
               end
cfWriteB4b:    begin //write b4
                cfWriteDone<=1'b0;
                cfState<=cfWriteComDone;
                microData<=8'h01;
                microWrite<=1'b1;
               end
cfWriteComDone: begin //do nothing for one cycle
                cfState<=cfWriteWaitBusy;
                cfWriteDone<=1'b0;
                microWrite<=1'b1;
               end
cfWriteWaitBusy:begin //wait for busy to clear
                cfState<=(cfBusy==1'b0) ? cfWriteEnd0 :
                   cfWriteWaitBusy;
                cfWriteDone<=1'b0;
                microWrite<=1'b1;
               end
cfWriteEnd0:   begin //increment lba, clear cf enable
                cfState<=cfWriteEnd1;
```

```
                cfLBA<=cfLBA+1'b1;
                microWrite<=1'b1;
                cfWriteDone<=1'b1;
            end
        cfWriteEnd1:  begin //do nothing for one cycle
                cfState<=cfWriteWaiting;
                microWrite<=1'b1;
                cfWriteDone<=1'b1;
            end
        default:    begin
                cfState<=cfWriteWaiting;
            end
    endcase
  end
end

//send prep envelopes to wifi controller and cf controller
always @(posedge clk) begin
  oldPrepNewEnvelope<=prepNewEnvelope;
  if(reset) begin
    pendingEnv<=1'b0;
    pendingCfWrite<=1'b0;
    etherState<=etherWait;
    etherBufferWr<=1'b0;
    cfWriteBufferAddrIn<=8'b0;
```

```
    cfWriteBufferWr<=1'b0;
    envCount<=3'b0;
    initDone<=1'b0;
  end
  else begin
    if(initCount<initPer && initDone==1'b0) begin //do
        initializations, setup cfWriteBuffer for test
      cfWriteBufferAddrIn<=(initCount[2:0]==3'b111) ?
          cfWriteBufferAddrIn+1'b1 : cfWriteBufferAddrIn;
      cfWriteBufferDataIn<={8'hf0,initCount[10:3]};
      cfWriteBufferWr<=(initCount[2:0]==3'b100);
      initDone<=(initCount[2:0]==3'b111 && cfWriteBufferAddrIn
          ==8'hff);
    end
    else begin
      etherBufferWr<=(etherState==etherWrite1 || etherState==
          etherWrite3);
      cfWriteBufferWr<=(etherState==etherWrite1);
      if(prepNewEnvelope==1'b1 && oldPrepNewEnvelope==1'b0 &&
          pendingEnv==1'b0) begin //if a new set of envelopes
          just arrived, start send process
        pendingEnv<=1'b1;
        etherState<=etherWait;
        envCount<=3'b0;
      end
```

```
else begin
  case(etherState)
    etherWait:   begin
                   etherState<=(pendingEnv==1'b1 &&
                     pendingCfWrite==1'b0) ? etherGet :
                     etherWait;
                   pendingCfWrite<=cfWriteDone ? 1'b0 :
                     pendingCfWrite;
                 end
    etherGet:    begin
                   {env0,env1,env2,env3,env4,env5,env6,env7}<=
                     prepEnvelopes;
                   etherState<=etherWrite0;
                 end
    etherWrite0:begin
                   etherState<=etherWrite1;
                   case(envCount)
                     3'b000: begin
                               etherBufferDataIn<=env0[15:8];
                               cfWriteBufferDataIn<=env0;
                             end
                     3'b001: begin
                               etherBufferDataIn<=env1[15:8];
                               cfWriteBufferDataIn<=env1;
                             end
```

```
        3'b010: begin
                etherBufferDataIn<=env2[15:8];
                cfWriteBufferDataIn<=env2;
            end
        3'b011: begin
                etherBufferDataIn<=env3[15:8];
                cfWriteBufferDataIn<=env3;
            end
        3'b100: begin
                etherBufferDataIn<=env4[15:8];
                cfWriteBufferDataIn<=env4;
            end
        3'b101: begin
                etherBufferDataIn<=env5[15:8];
                cfWriteBufferDataIn<=env5;
            end
        3'b110: begin
                etherBufferDataIn<=env6[15:8];
                cfWriteBufferDataIn<=env6;
            end
        3'b111: begin
                etherBufferDataIn<=env7[15:8];
                cfWriteBufferDataIn<=env7;
            end
    endcase
```

```
        end
etherWrite1 : begin
            etherState<=etherWrite2;
        end
etherWrite2 : begin
            etherState<=etherWrite3;
            cfWriteBufferAddrIn<=cfWriteBufferAddrIn+1'
                b1;
            case(envCount)
              3'b000: etherBufferDataIn<=env0[7:0];
              3'b001: etherBufferDataIn<=env1[7:0];
              3'b010: etherBufferDataIn<=env2[7:0];
              3'b011: etherBufferDataIn<=env3[7:0];
              3'b100: etherBufferDataIn<=env4[7:0];
              3'b101: etherBufferDataIn<=env5[7:0];
              3'b110: etherBufferDataIn<=env6[7:0];
              3'b111: etherBufferDataIn<=env7[7:0];
            endcase
        end
etherWrite3 : begin
            envCount<=envCount+1'b1;
            etherState<=(envCount==maxEnv) ? etherDone :
                 etherWrite0;
        end
etherDone:   begin
```

```
                    etherState<=etherWait;
                    pendingCfWrite<=(cfWriteBufferAddrIn==8'h00)
                        ;
                    pendingEnv<=1'b0;
                  end

          endcase
        end
      end
    end
  end


  always @(posedge clk) begin
    cfReadBufferAddrOut<=cfReadBufferAddrOut+1'b1;
  end
endmodule
```

```
module prep(clk,reset,v,i,newSample,sendRaw,envelopes,
   newEnvelope,led);
  input clk;           //system clk, 25MHz
  input reset;         //global reset, active high
  input[7:0] v;        //voltage sample
  input[7:0] i;        //curent sample
  input newSample;       //high for one cycle when a new sample
     of v and i arrives
  input sendRaw;         //used to force sending raw data instead
      of prep data

  output[127:0] envelopes;     //spectral envelopes produced by
     prep
  output newEnvelope;      //high for one cycle when a new
     envelope is produced
  output led;
  reg led;

  wire[127:0] envelopes;
  reg newEnvelope;
  reg[6:0] basisAddr;          //used to index basis sinusoids
  reg squareV;             //square wave with same frequency and
     phase as v
  reg oldSquareV;            //squareV delayed by one sample
  reg delSquareV;            //squareV delayed by one clock cycle
```

```
reg[4:0] debounceCount;        //used to debounce v
reg oldNewSample;             //newSample delayed by one clock
   cycle
reg[2:0] envState;             //state for envelope fsm
reg signed[22:0] envAcc0,envAcc1, //accumulators for envelopes
    envAcc2,envAcc3,envAcc4,
    envAcc5,envAcc6,envAcc7;
reg[2:0] envNum;              //current envelope being worked on
reg[6:0] rawCount;             //used to count packets for raw
   transmission
reg squareVSync;              //sync signal produced by cleaned v
reg[23:0] numCycles;           //period of squareV in cycles
reg[23:0] cycleCount;          //used to produce numCycles
reg[23:0] addrCycleCount;      //used to count time between
   address increments
reg envFired;                 //used to signify envelope
   transmission
reg[127:0] iRaw;              //stores raw i



//envelope creation fsm state enum
parameter wait0=3'b000;
parameter wait1=3'b001;
parameter getV=3'b010;
```

```
parameter waitMult0=3'b011;
parameter waitMult1=3'b100;
parameter getProd=3'b101;
parameter done=3'b110;

parameter numEnvelopes=4'h8; //number of envelopes to compute

//basisRom module and connections
wire[63:0] basisOut;
basisRom aBasisRom(basisAddr,~clk,basisOut); //~clk used to
   meet setup and hold times of ROM

//prepMult module and connections
reg signed[7:0] multIn0,multIn1; //multIn0 is i, multIn1 is a
   basis sinusoid
wire signed[15:0] multOut;
prepMult aPrepMult(~clk,multIn0,multIn1,multOut);

assign envelopes=(sendRaw==1'b1) ? iRaw : {envAcc0[19:4],
   envAcc1[19:4],envAcc2[19:4],envAcc3[19:4],envAcc4[19:4],
   envAcc5[19:4],envAcc6[19:4],envAcc7[19:4]};

//generate addr to basis sinusoid rom
always @(posedge clk) begin
  delSquareV<=squareV;
```

```
if(reset) begin
  oldNewSample<=1'b0;
  squareV<=1'b0;
  debounceCount<=5'b0;
  basisAddr<=7'b0;
  numCycles<=24'h065B9A;
  cycleCount<=24'b0;
end
else begin
  oldNewSample<=newSample;
  //produce squareV
  if(newSample==1'b1 && oldNewSample==1'b0) begin //if a new
      sample was received, process it
    oldSquareV<=squareV;
    //produce squareV by passing v through a hysteretic
       comparator, after detecting an edge, won't
    //encounter a new edge for at least 32 samples
    if(squareV==1'b0 && v>8'h7f && debounceCount==5'b0)
       begin //rising edge
      squareV<=1'b1;
      debounceCount<=5'b00001;
    end
    else if(squareV==1'b1 && v<8'h80 && debounceCount==5'b0)
        begin //falling edge
      squareV<=1'b0;
```

```
    debounceCount<=5'b00001;
  end
  else begin //no edge
    debounceCount<=(debounceCount==5'b0) ? 5'b0 :
       debounceCount+1'b1;
  end
end

//produce basisAddr using squareV
if(squareV==1'b1 && delSquareV==1'b0) begin //if there is
   a rising edge in squareV, reset basisAddr, assures sync
  basisAddr<=7'b0;
end
else begin
  //basisAddr<=(newSample==1'b1 && oldNewSample==1'b0) ?
     basisAddr+1'b1 : basisAddr;
  basisAddr<=(addrCycleCount==((numCycles>>4'h7)-1'b1)) ?
     basisAddr+1'b1 : basisAddr;
end

//produce sync pulse
squareVSync<=(squareV!=delSquareV);

//produce cycle counts
```

```
    cycleCount<=(squareV==1'b1 && delSquareV==1'b0) ? 24'b0 :
        cycleCount+1'b1;
    numCycles<=(squareV==1'b1 && delSquareV==1'b0) ? ((
        cycleCount+numCycles)>>1'b1) : numCycles;
    addrCycleCount<=((addrCycleCount==((numCycles>>4'h7)-1'b1)
        ) || (squareV==1'b1 && delSquareV==1'b0)) ? 24'b0 :
        addrCycleCount+1'b1;
  end
end

//produce envelopes
always @(posedge clk) begin
  led<=sendRaw;
  if(reset) begin
    newEnvelope<=1'b0;
    envState<=done;
    envNum<=3'b0;
    rawCount<=7'b0;
    envFired<=1'b0;
    envAcc0<=23'sb0;
    envAcc1<=23'sb0;
    envAcc2<=23'sb0;
    envAcc3<=23'sb0;
    envAcc4<=23'sb0;
    envAcc5<=23'sb0;
```

```
    envAcc6<=23'sb0;
    envAcc7<=23'sb0;
  end
  else begin
    if(newSample==1'b1 && oldNewSample==1'b0) begin //if a new
        sample was received, process it
      multIn0<=(i>8'h7f) ? (i-8'h80) : ((~(8'h80-i))+1'b1);//
          convert to signed, two's complement
      envState<=wait0;
      envNum<=3'b0;
      rawCount<=rawCount+1'b1;
      newEnvelope<=1'b0;
    end
    else begin
      if(sendRaw==1'b0) begin//do normal prep data
        newEnvelope<=(basisAddr==7'b0 && squareVSync==1'b1);
            //if we just cycled to data for next envelope, send
             current envelope
        case(envState)
          wait0: envState<=wait1;
          wait1: envState<=getV;
          getV:    begin
                   case(envNum)
                     3'b000: multIn1<=basisOut[63:56];
                     3'b001: multIn1<=basisOut[55:48];
```

```
            3'b010: multIn1<=basisOut[47:40];
            3'b011: multIn1<=basisOut[39:32];
            3'b100: multIn1<=basisOut[31:24];
            3'b101: multIn1<=basisOut[23:16];
            3'b110: multIn1<=basisOut[15:8];
            3'b111: multIn1<=basisOut[7:0];
          endcase
          envState<=waitMult0;
        end
waitMult0: envState<=waitMult1;
waitMult1: envState<=getProd;
getProd:  begin
            case(envNum)
              3'b000: envAcc0<=(basisAddr==7'b0 &&
                 oldSquareV!=squareV)? multOut : envAcc0
                 +multOut;
              3'b001: envAcc1<=(basisAddr==7'b0 &&
                 oldSquareV!=squareV)? multOut : envAcc1
                 +multOut;
              3'b010: envAcc2<=(basisAddr==7'b0 &&
                 oldSquareV!=squareV)? multOut : envAcc2
                 +multOut;
              3'b011: envAcc3<=(basisAddr==7'b0 &&
                 oldSquareV!=squareV)? multOut : envAcc3
                 +multOut;
```

```
                3'b100: envAcc4<=(basisAddr==7'b0 &&
                    oldSquareV!=squareV)? multOut : envAcc4
                    +multOut;
                3'b101: envAcc5<=(basisAddr==7'b0 &&
                    oldSquareV!=squareV)? multOut : envAcc5
                    +multOut;
                3'b110: envAcc6<=(basisAddr==7'b0 &&
                    oldSquareV!=squareV)? multOut : envAcc6
                    +multOut;
                3'b111: envAcc7<=(basisAddr==7'b0 &&
                    oldSquareV!=squareV)? multOut : envAcc7
                    +multOut;
              endcase
              envNum<=envNum+1'b1;
              envState<=(envNum==(numEnvelopes-1)) ? done
                  : wait1;
            end
      done: envState<=envState;
      default: envState<=envState;
    endcase
  end
  else begin //send raw data
    newEnvelope<=(rawCount==7'h7f && envFired==1'b0);
    envFired<=(rawCount==7'h0) ? 1'b0 : (rawCount==7'h7f ?
        1'b1 : envFired);
```

```
            case(rawCount[6:3])
              4'h0: iRaw[127:120]<=i;
              4'h1: iRaw[119:112]<=i;
              4'h2: iRaw[111:104]<=i;
              4'h3: iRaw[103:96]<=i;
              4'h4: iRaw[95:88]<=i;
              4'h5: iRaw[87:80]<=i;
              4'h6: iRaw[79:72]<=i;
              4'h7: iRaw[71:64]<=i;
              4'h8: iRaw[63:56]<=i;
              4'h9: iRaw[55:48]<=i;
              4'ha: iRaw[47:40]<=i;
              4'hb: iRaw[39:32]<=i;
              4'hc: iRaw[31:24]<=i;
              4'hd: iRaw[23:16]<=i;
              4'he: iRaw[15:8]<=i;
              4'hf: iRaw[7:0]<=i;
            endcase
          end
        end
      end
    end

endmodule
```

```
//this module controls the ADC
//when enabled, it samples both input channels at 7.68KHz,
   sequential sampling
//and stores the result in the buffer
module adcController(clk, reset, enable, adc_db, adc_cs, adc_rd,
   adc_wr, adc_intrq, adc_clk, bufferData, bufferWrite, bufferClk);
  input clk;            //system clk, 25MHz
  input reset;          //global reset, active high
  input enable;         //asserted to enable this module
  input adc_intrq;       //interrupt request from adc, active low

  inout[7:0] adc_db;       //databus to adc

  output adc_cs;         //chip select for adc, active low
  output adc_rd;         //read for adc, active low
  output adc_wr;         //write for adc, active low
  output adc_clk;        //clock for adc, has 1/16 freq of clk
  output bufferClk;      //clock for fifo buffer, inverted
     adc_clk
  output[15:0] bufferData;     //data to be written in fifo
     buffer
  output bufferWrite;      //write signal for fifo buffer, active
      high

  wire[7:0] adc_db;
```

```
wire adc_cs, adc_clk, bufferClk;
reg bufferWrite, adc_rd, adc_wr;
reg[15:0] bufferData;

reg[7:0] dataReg;      //data to be written to adc_db
reg adcDatabusWrite;    //asserted to drive the adc databus
   with dataReg
reg[3:0] outClkCount;   //used to produce output clock to adc
reg[2:0] state;        //state for fsm
reg chan;          //stores next channel to be read from
reg[11:0] sampleClkCount; //used to generate sample clk

//config params, used to select between adc channels and setup
    sampling mode
parameter chan0Config=8'b10100100; //left justified, 10-bit,
   unsigned, single ended, chan 0
parameter chan1Config=8'b10100101; //left justified, 10-bit,
   unsigned, single ended, chan 1

//fsm state enum
parameter setupConfig=3'b000;
parameter writeConfig=3'b001;
parameter waitForInt=3'b010;
parameter firstRead=3'b011;
parameter pauseRead=3'b100;
```

```
parameter secondRead=3'b101;
parameter nop=3'b110;

//number of clk cycles between samples
parameter numCycles=1628;

assign adc_db=adcDatabusWrite ? dataReg : 8'hZZ;
assign adc_clk=outClkCount[3];
assign adc_cs=~enable;
assign bufferClk=~clk; //inverted to meet setup and hold times
    of buffer



//interact with adc
always @(posedge clk) begin
    outClkCount<=outClkCount+1'b1;
  bufferWrite<=(enable==1'b1 && state==secondRead && chan==1'
     b1 && outClkCount==4'b0011);
  adcDatabusWrite<=(enable==1'b1 && (state==setupConfig ||
     state==writeConfig));
  adc_wr<=~(enable==1'b1 && state==writeConfig);
  adc_rd<=~(enable==1'b1 && (state==firstRead || state==
     secondRead));
```

```
if(reset) begin
  chan<=1'b0;
  state<=nop;
  dataReg<=8'b01011011;
  sampleClkCount<=12'b0;
end
else if(enable) begin
  sampleClkCount<=(sampleClkCount==numCycles) ? 12'b0 :
     sampleClkCount+1'b1;
  case(state)
    setupConfig:     begin //load config word to databus
                dataReg<=chan ? chan1Config : chan0Config;
                state<=(outClkCount==4'b0100 && sampleClkCount
                   <{8'b0,5'b11111}) ? writeConfig :
                   setupConfig;
              end
    writeConfig:  begin //write config word to adc
                state<=(outClkCount==4'b0100) ? waitForInt :
                   writeConfig;
              end
    waitForInt:    begin //wait until int strobes low,
       indicates conversion is finished
                state<=(outClkCount==4'b0100) ? (adc_intrq ?
                   waitForInt : firstRead) : waitForInt;
              end
```

```
    firstRead:      begin //read high byte
                    bufferData<=chan ? {adc_db,bufferData[7:0]} :
                        {bufferData[15:8],adc_db};
                    state<=(outClkCount==4'b0100) ? pauseRead :
                        firstRead;
                end
    pauseRead:      begin //pause between reads
                    state<=(outClkCount==4'b0100) ? secondRead :
                        pauseRead;
                end
    secondRead:     begin //read low 2 bits, write high byte
       to buffer when both channels are done
                    state<=(outClkCount==4'b0100) ? setupConfig :
                        secondRead;
                    chan<=(outClkCount==4'b0100) ? ~chan : chan;
                end
    nop:            begin
                    state<=(outClkCount==4'b0100) ? setupConfig :
                        nop;
                end
  endcase
end
else begin
  state<=nop;
  dataReg<=8'b01011011;
```

```
    end
  end
endmodule
```

```
//this module controls the cf card
//when issuing a load buffer command, data should be of the form
//byte0: low 8 bits of sector address
//byte1: next higher 8 bits of sector address
//byte2: next higher 8 bits of sector address
//byte3: 4 zeros, high 4 bits of sector address
//byte4: number of sectors to read
//
//when issuing a write sector command, data should be of the
   form
//byte0: low 8 bits of sector address
//byte1: next higher 8 bits of sector address
//byte2: next higher 8 bits of sector address
//byte3: 4 zeros, high 4 bits of sector address
//byte4: the literal 01h
module cfController(clk,reset,readEnable,microWrite,microData,
   cs0, cs1, addr, cfData,read,write,cfReg,cfReset,intrq,debug,
   bufferData,bufferAddr,bufferWr,busy,isReadMode,
   writeBufferAddr, writeBufferData,led);
  input clk;  //system clk
  input reset;  //global reset;
  input readEnable;   //asserted to enable read data routine
  input microWrite;   //write line from micro, active low
  input[7:0] microData; ///micro databus
  input intrq;      //interrupt request
```

```
input isReadMode;      //1 for read mode, 0 for write mode
input[15:0] writeBufferData; //write buffer ram data bus


inout [15:0] cfData;  //databus to compact flash

output cs0;        //used to access task file, active low
output cs1;        //used to access alternate status register
   and device control register, active low
output [2:0] addr;     //address
output read;       //read
output write;      //write
output cfReg;      //not used, should be always a logic 1
output cfReset;      //reset, active low
output[15:0] bufferData; //buffer ram data bus
output[7:0] bufferAddr; //buffer ram address
output[7:0] writeBufferAddr; //write buffer ram address
output bufferWr;     //buffer ram write, active high
output busy;       //asserted when device is busy, active high
output[3:2] debug;
output led;
reg led;

reg busy;
reg[3:2] debug;
```

```
reg oldWrite; //write delayed by 1 clock cycle
reg oldReadEnable; //readEnable delayed by 1 clock cycle
wire cfReg,cfReset;
wire[15:0] cfData;
reg[2:0] addr;
reg[7:0] bufferAddr;
wire[7:0] writeBufferAddr;
reg[15:0] cfDataReg,bufferData;
reg read,write,cs0,cs1,bufferWr;
reg cfDatabusWrite; //asserted to write to cf databus, active
   high
reg[4:0] cfCmd; //stores current command
reg[4:0] oldCfCmd; //stores command at previous cycle
reg[27:0] lba; //stores logical block address
reg[7:0] sectorCount; //stores sector count
reg[9:0] cfCount; //temporary
reg[19:0] initCount; //used to achieve pause at powerup
reg[6:0] cfState; //used to maintain state information of FSM
reg tempBusy; //temporarily stores busy bit
reg[2:0] packetCount; //counts data packets
reg pendingRead; //asserted when there is a pending read
   request
reg[7:0] wordCount; //used to count words during transfer
   cycle
reg[7:0] numSectorsRead;  //counts number of sectors read
```

```
assign cfReset=~reset;
assign cfReg=1'b1;
assign cfData= (cfDatabusWrite==1'b1) ? cfDataReg : 16'hZZZZ;
assign writeBufferAddr=bufferAddr;

parameter initPer=50000; //the device remains innactive for
   initPer cycles after reset
parameter readSectorCmd=8'h20; //read with retries
parameter writeSectorCmd=8'h30; //write with retries
parameter readBufferCmd=8'he4; //read buffer


//enumeration of commands, code 00010 currently unused, was
   previously read alt status
parameter nop=5'b00000;
parameter initCard=5'b00001;
parameter readStatus=5'b00011;
parameter readTest=5'b00100;
parameter read0=5'b00101; //wait for busy to clear
parameter read1=5'b00110; //write to card/head register
parameter read2=5'b00111; //wait for busy to clear and drdy to
    assert
parameter read3=5'b01000; //send cylinder high
parameter read4=5'b01001; //send cylinder low
```

```
parameter read5=5'b01010; //send starting sector
parameter read6=5'b01011; //send number of sectors
parameter read7=5'b01100; //write command code (read sector)
parameter read8=5'b01101; //wait for busy to clear and drq to
   set
parameter read9=5'b01110; //read a block of data

always @(posedge clk) begin
  led<=busy;
  oldWrite<=microWrite;
  oldReadEnable<=readEnable;
  initCount<=(initCount<initPer) ? initCount+1'b1 : initCount;
  if(reset)
    initCount<=20'b0;
  if(reset || (initCount<initPer)) begin
    cfDatabusWrite<=1'b0;
    cs0<=1'b1;
    cs1<=1'b1;
    read<=1'b1;
    write<=1'b1;
    debug<=2'b0;
    lba<=28'hffffffff;
    cfDataReg<=16'hffff;
    cfCount<=10'b0;
    cfCmd<=initCard;
```

```
    oldCfCmd<=nop;
    addr<=3'b110;
    tempBusy<=1'b1;
    bufferWr<=1'b0;
    bufferData<=16'b0;
    bufferAddr<=8'b0;
    packetCount<=3'b0;
    pendingRead<=1'b0;
    wordCount<=8'b0;
    busy<=1'b1;
    numSectorsRead<=8'b0;
  end
  //process command
  else begin

    //if module is in read mode, start gathering lba data from
        micro
    if(readEnable) begin
      //if readEnable just stepped high, reset packet count
      if(oldReadEnable==1'b0) begin
        packetCount<=2'b0;
        pendingRead<=1'b0;
      end
      //if write just stepped low, grab a data packet
      else if((oldWrite==1'b1) && (microWrite==1'b0)) begin
```

```
      case(packetCount)
        3'b000: lba<={lba[27:8],microData};
        3'b001: lba<={lba[27:16],microData,lba[7:0]};
        3'b010: lba<={lba[27:24],microData,lba[15:0]};
        3'b011: lba<={microData[3:0],lba[23:0]};
        default: sectorCount<=microData;
      endcase
      packetCount<=(packetCount==3'b100) ? 3'b100 :
          packetCount+1'b1;
      pendingRead<=(packetCount==3'b100);
    end
  end

  oldCfCmd<=cfCmd;
  //when command just changed, reset state and tempBusy
  if(~(cfCmd==oldCfCmd)) begin
    cfState<=7'b0;
    tempBusy<=1'b1;
    busy<=1'b1;
  end
  //otherwise, process current command
  else begin
    cfState<=cfState+1'b1;
    case(cfCmd)
```

```
initCard:begin //start initializing compact flash card
    by
      //reading status register (not alt status)
      //nb: this is functionally the same as
         readStatus
      //however upon completion of this command,
         readTest is
      //called, not nop
        cs0<=1'b0;
        cs1<=1'b1;
        addr<=3'b111;
        read<=(cfState[6]==cfState[5]); //asserted for
            01,10 as high bits
        write<=1'b1;
        cfDatabusWrite<=1'b0;
        cfDataReg<=16'b0;
        cfCmd<=(cfState==7'b1111111 && debug[2]==1'b1)
            ? readTest : cfCmd; //enters readTest at
           end
        debug[2]<=(cfState==7'b1011111) ? (~cfData[7]
           && cfData[6] && cfData[4]) : debug[2]; //
           checks that ready and dsc are set, and busy
            is cleared
        busy<=1'b1;
    end
```

```
readStatus: begin //reads status register (not alt
   status)
        cs0<=1'b0;
        cs1<=1'b1;
        addr<=3'b111;
        read<=(cfState[6]==cfState[5]); //asserted for
            01,10 as high bits
        write<=1'b1;
        cfDatabusWrite<=1'b0;
        cfDataReg<=16'b0;
        cfCmd<=(cfState==7'b1111111) ? nop : cfCmd; //
           enters nop at end
        busy<=1'b1;
      end
readTest: begin //performs a test read
        lba<=28'h0000100;
        cfCmd<=read0;
        sectorCount<=8'h01;
        busy<=1'b1;
      end
read0:    begin //wait for busy to clear, functionally
    the same as readStatus
        cs0<=1'b0;
        cs1<=1'b1;
        addr<=3'b111;
```

```
            read<=(cfState[6]==cfState[5]); //asserted for
                01,10 as high bits
            write<=1'b1;
            cfDatabusWrite<=1'b0;
            cfDataReg<=16'b0;
            cfCmd<=(cfState==7'b1111111 && tempBusy==1'b0)
                ? read1 : cfCmd; //enters read1 at end
            tempBusy<=(cfState==7'b1011111) ? cfData[7] :
               tempBusy;
            busy<=1'b1;
        end
read1:    begin //write to card/head reg
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b110;
            read<=1'b1;
            write<=(cfState[6]==cfState[5]); //asserted
               for 01,10 as high bits
            cfDatabusWrite<=1'b1;
            cfDataReg<={8'b0,4'b1110,lba[27:24]};
            cfCmd<=(cfState==7'b1111111) ? read2 : cfCmd;
               //enters read2 at end
            busy<=1'b1;
        end
```

```
read2:    begin //wait for busy to clear, and ready to
    assert
        cs0<=1'b0;
        cs1<=1'b1;
        addr<=3'b111;
        read<=(cfState[6]==cfState[5]); //asserted for
            01,10 as high bits
        write<=1'b1;
        cfDatabusWrite<=1'b0;
        cfDataReg<=16'b0;
        cfCmd<=(cfState==7'b1111111 && tempBusy==1'b0)
            ? read3 : cfCmd; //enters read3 at end
        tempBusy<=(cfState==7'b1011111) ? (cfData[7] |
            ~cfData[6]) : tempBusy;
        busy<=1'b1;
      end
read3:    begin //write cylinder high reg
        cs0<=1'b0;
        cs1<=1'b1;
        addr<=3'b101;
        read<=1'b1;
        write<=(cfState[6]==cfState[5]); //asserted
           for 01,10 as high bits
        cfDatabusWrite<=1'b1;
        cfDataReg<={8'b0,lba[23:16]};
```

```
            cfCmd<=(cfState==7'b1111111) ? read4 : cfCmd;
               //enters read4 at end
            busy<=1'b1;
         end
read4:     begin //write cylinder low reg
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b100;
            read<=1'b1;
            write<=(cfState[6]==cfState[5]); //asserted
               for 01,10 as high bits
            cfDatabusWrite<=1'b1;
            cfDataReg<={8'b0,lba[15:8]};
            cfCmd<=(cfState==7'b1111111) ? read5 : cfCmd;
               //enters read5 at end
            busy<=1'b1;
         end
read5:     begin //write sector number reg
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b011;
            read<=1'b1;
            write<=(cfState[6]==cfState[5]); //asserted
               for 01,10 as high bits
            cfDatabusWrite<=1'b1;
```

```
            cfDataReg<={8'b0,lba[7:0]};
            cfCmd<=(cfState==7'b1111111) ? read6 : cfCmd;
               //enters read6 at end
            busy<=1'b1;
         end
read6:      begin //write sector count reg
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b010;
            read<=1'b1;
            write<=(cfState[6]==cfState[5]); //asserted
               for 01,10 as high bits
            cfDatabusWrite<=1'b1;
            cfDataReg<=(isReadMode==1'b1) ? {8'b0,
               sectorCount} : {15'b0,1'b1};
            cfCmd<=(cfState==7'b1111111) ? read7 : cfCmd;
               //enters read7 at end
            busy<=1'b1;
         end
read7:      begin //write read sector command
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b111;
            read<=1'b1;
```

```
            write<=(cfState[6]==cfState[5]); //asserted
               for 01,10 as high bits
            cfDatabusWrite<=1'b1;
            cfDataReg<=(isReadMode==1'b1) ? {8'b0,
               readSectorCmd} : {8'b0,writeSectorCmd};
            cfCmd<=(cfState==7'b1111111) ? read8 : cfCmd;
               //enters read8 at end
            bufferAddr<=8'b0;
            busy<=1'b1;
            numSectorsRead<=8'b0;
         end
read8:     begin //read alt status until busy is
   cleared and drq is set
            //when reading, also wait for interrupt, when
               doing a write, do not wait
            cs0<=1'b1;
            cs1<=1'b0;
            addr<=3'b110;
            read<=(cfState[6]==cfState[5]); //asserted for
                01,10 as high bits
            write<=1'b1;
            cfDatabusWrite<=1'b0;
            cfDataReg<=16'b0;
            cfCmd<=(cfState==7'b1111111 && tempBusy==1'b0
               && (intrq==1'b1 || isReadMode==1'b0)) ?
```

```
                read9 : cfCmd; //enters read9 at end
            tempBusy<=(cfState==7'b1011111) ? (cfData[7] |
                ~cfData[3]) : tempBusy;
            wordCount<=8'b0;
            busy<=1'b1;
        end
read9:      begin //read or write a block of data
            cs0<=1'b0;
            cs1<=1'b1;
            addr<=3'b000;
            read<=(isReadMode==1'b0 || cfState[6]==cfState
                [5]); //asserted for 01,10 as high bits, in
                 read mode
            write<=(isReadMode==1'b1 || cfState[6]==
                cfState[5]); //asserted for 01,10 as high
                bits, in write mode
            cfDatabusWrite<=~isReadMode;
            cfDataReg<=(cfState==7'b0000100) ?
                writeBufferData : cfDataReg;
            numSectorsRead<=(cfState==7'b1111110 &&
                wordCount==8'b0) ? numSectorsRead+1'b1 :
                numSectorsRead;
            wordCount<=(cfState==7'b1111101) ? wordCount
                +1'b1 : wordCount;
```

```
            cfCmd<=(cfState==7'b1111111 && wordCount==8'b0
                ) ? ((numSectorsRead>=sectorCount) ?
                readStatus : read8) : cfCmd; //enters
                readStatus or read8 at end
            bufferData<=(cfState==7'b1011111) ? cfData :
                bufferData;
            bufferWr<=(isReadMode==1'b1 && cfState==7'
                b1100000);
            bufferAddr<=(cfState==7'b1111111) ? bufferAddr
                +1'b1 : bufferAddr;
            busy<=1'b1;
        end
    default:  begin //includes nop
            cs0<=1'b1;
            cs1<=1'b1;
            addr<=3'b0;
            read<=1'b1;
            write<=1'b1;
            cfDatabusWrite<=1'b0;
            cfDataReg<=16'b0;
            debug[3]<=1'b1;
            if(pendingRead) begin //service read request
              cfCmd<=read0;
              pendingRead<=1'b0;
            end
```

```
                busy<=1'b0;

            end
        endcase
      end
    end
  end
endmodule

//reads two bytes of data from compact flash buffer, high byte
   first
//input data should be of the form
//byte 0: word number
module cfBufferReader(clk, reset, enable, microData, dataReg,
   write,read, bufferAddr, bufferData);
  input clk;     //system clk
  input reset;   //global reset
  input enable; //1 when module is active
  input write;   //write signal from microcontroller, active low
  input read;    //read signal from microcontroller, active low
  input[15:0] bufferData; //buffer data out
  input[7:0] microData; //microcontroller databus

  output[7:0] dataReg; //output register
  output[7:0] bufferAddr; //buffer read addr line
```

```
reg[7:0] bufferAddr;
reg oldWrite; //write delayed by one cycle
reg oldRead;  //read delayed by one cycle
reg[7:0] dataReg; //stores data to be driven on bus
reg highByte; //one when high byte of word is being requested
reg oldEnable;  //enable delayed by one cycle

always @(posedge clk) begin
  oldWrite<=write;
  oldRead<=read;
  oldEnable<=enable;
  if(reset) begin
    highByte<=1'b1;
    dataReg<=8'b0;
  end
  else begin
    if(enable) begin
      //if enable just stepped high, reset highByte
      if(oldEnable==1'b0) begin
        highByte<=1'b1;
      end
      //if write just stepped low, read a packet of data
      else if((write==1'b0) && (oldWrite==1'b1)) begin
        bufferAddr<=microData;
```

```
        end
        //otherwise, if read just stepped low, load dataReg
           appropriately
        else if((read==1'b0) && (oldRead==1'b1)) begin
          dataReg<=highByte ? bufferData[15:8] : bufferData
             [7:0];
          highByte<=~highByte;
        end
      end
    end
  end
endmodule

//writes two bytes of data to compact flash buffer, high byte
   first
//input data should be of the form
//byte 0: word number
//byte 1: high data byte
//byte 2: low data byte
module cfBufferWriter(clk, reset, enable, microData, microWrite,
    bufferAddr, bufferData,bufferWrite);
  input clk;     //system clk
  input reset;   //global reset
  input enable; //1 when module is active
  input[7:0] microData; //microcontroller databus
```

```
input microWrite; //write signal from microcontroller, active
   low


output[7:0] bufferAddr; //buffer read addr line
output[15:0] bufferData; //buffer data out
output bufferWrite;    //buffer write line, active high

reg[7:0] bufferAddr;
reg[15:0] bufferData;
reg bufferWrite;
reg oldWrite; //microWrite delayed by one cycle
reg oldEnable;  //enable delayed by one cycle
reg pendingWrite; //1 when write to buffer is pending
reg[1:0] packetCount; //counts input packets

always @(posedge clk) begin
  oldWrite<=microWrite;
  oldEnable<=enable;
  if(reset) begin
    pendingWrite<=1'b0;
    bufferWrite<=1'b0;
  end
  else begin
    if(enable) begin
```

```
    //if enable just stepped high, reset packetCount and
       pendingWrite
    if(oldEnable==1'b0) begin
      packetCount<=2'b0;
      pendingWrite<=1'b0;
    end
    //if write just stepped low, read a packet of data
    else if((microWrite==1'b0) && (oldWrite==1'b1)) begin
      packetCount<=(packetCount==2'b11) ? 2'b11 :
         packetCount+1'b1;
      case(packetCount)
        2'b00: bufferAddr<=microData;
        2'b01: bufferData<={microData,bufferData[7:0]};
        2'b10: bufferData<={bufferData[15:8],microData};
        default: bufferData<=bufferData;
      endcase
      pendingWrite<=(packetCount==2'b10);
    end
    else begin
      pendingWrite<=1'b0;
    end
    bufferWrite<=pendingWrite;
  end
  else begin
    bufferWrite<=1'b0;
```

```
        end
      end
    end
endmodule
```

```
module etherController(clk,reset,rxd,txd,modReset,fifoRead,
  fifoData,fifoEmpty,fifoUsedWords,sendRaw,led);
  input clk;            //system clk, 25MHz
  input reset;          //global reset,active high
  input rxd;            //serial input line from wifi module
  input[7:0] fifoData;     //databus from fifo
  input fifoEmpty;       //high when fifo is empty
  input[9:0] fifoUsedWords; //number of data words in fifo

  output txd;           //serial output line to wifi module
  output fifoRead;       //read line for fifo, active high
  output modReset;       //reset line for wifi module, active low
  output sendRaw;        //used to force sending raw data instead
      of prep data
  output led;


  reg led;
  reg fifoRead;
  reg sendRaw;
  reg[2:0] state;        //state for transmit fsm
  reg[31:0] initCount;     //used to keep module inactive for a
    designated amount of time at startup
  reg [2:0] cmd;         //used to select current command
```

```
reg[7:0] byteCount;      //used to count bytes in send/recv
   stream
reg[19:0] waitCount;     //used to wait during recv
reg sendingDataPacket;     //used to keep track of packet type
reg commandReceived;     //used to signify a command has been
   received via wifi
reg[7:0] command;      //stores command from wifi
reg justSentStatus;      //used to record sending of status
   updates

wire modReset;

parameter initPer=375000000;
parameter maxByteCount=8'h81; //max value of byteCount
parameter lastInitAddr=7'h61; //last addr of init rom that
   stores init params
parameter readStart=7'h62; //first addr of read command
parameter readEnd=7'h6C; //last addr of read command
parameter preambleStart=7'h6D; //first addr of preamble
parameter preambleEnd=7'h7C; //last addr of preamble
parameter etherBaud=38400;


//transmit state enum
parameter waiting=3'b000;
```

```
parameter readRq=3'b001;
parameter moveData=3'b010;
parameter sendData=3'b011;
parameter doneData=3'b100;

//receive state enum
parameter rWaiting=3'b100;
parameter rMoveData=3'b101;
parameter rWriteData=3'b110;

//command enum
parameter doInit=3'b000;
parameter doReadCmd=3'b001;
parameter doReadData=3'b010;
parameter doPreamble=3'b011;
parameter doSendData=3'b100;

//header enum
parameter header_null=8'hff;
parameter header_data=8'h00;
parameter header_status=8'h01;
parameter header_debug=8'h02;

//receive command enum
parameter command_start=8'h43;
```

```
parameter command_transmit_no=8'h30;
parameter command_transmit_yes=8'h31;
parameter command_data_prep=8'h32;
parameter command_data_raw=8'h33;
parameter command_null=8'hf;

//async_receiver module and connections
wire rxDataReady,rxDebug,rxIdle;
wire[7:0] rxData;
async_receiver aAsync_receiver(clk,rxd,reset,rxDataReady,
   rxData,rxIdle,rxDebug);
defparam aAsync_receiver.Baud=etherBaud;

//async_transmitter module and connections
reg txStart;
wire txBusy;
reg[7:0] txData;
async_transmitter aAsync_transmitter(clk, txStart, txData, txd
   , txBusy);
defparam aAsync_transmitter.Baud=etherBaud;

//etherInit module and connections
reg[6:0] etherIAddr;
wire[7:0] etherIData;
etherInit aEtherInit(etherIAddr,~clk,etherIData);
```

```
//etherRecvFIFO module and connections
reg[7:0] rfifoIn;
reg rfifoWrite,rfifoRead,rfifoClr;
wire[7:0] rfifoOut;
wire rfifoEmpty;
etherRecvFIFO aEtherRecvFIFO(rfifoIn,rfifoWrite,rfifoRead,clk,
   rfifoClr,rfifoOut,rfifoEmpty);
assign modReset=~reset;

always @(posedge clk) begin
  initCount<=(initCount<initPer) ? initCount+1'b1 : initCount;
  if(reset || initCount<initPer) begin
    state<=waiting;
    fifoRead<=1'b0;
    rfifoRead<=1'b0;
    rfifoWrite<=1'b0;
    rfifoClr<=1'b1;
    rfifoIn<=8'h55;
    txStart<=1'b0;
    etherIAddr<=7'b0;
    cmd<=doInit;
    byteCount<=8'b0;
    led<=1'b0;
    sendingDataPacket<=1'b0;
```

```
    commandReceived<=1'b0;
    command<=command_null;
    sendRaw<=1'b0;
    justSentStatus<=1'b1;
  end
  else begin
    led<=(cmd==doPreamble);
    case(cmd)
      doInit: begin //initialize
            fifoRead<=1'b0;
            rfifoRead<=1'b0;
            rfifoWrite<=1'b0;
            rfifoClr<=1'b0;
            txData<=(state==moveData) ? etherIData : txData;
            txStart<=(state==sendData);
            etherIAddr<=(state==waiting && etherIAddr>
               lastInitAddr) ? readStart : ((state==moveData)
               ? etherIAddr+1'b1 : etherIAddr);
            cmd<=(state==waiting && etherIAddr>lastInitAddr) ?
                doReadCmd : cmd;
            case(state)
              waiting:  state<=(txBusy==1'b0) ? moveData :
                 waiting;
              moveData: state<=sendData;
```

```
            sendData:    state<=(txBusy==1'b1) ? waiting :
                sendData;
            default:   state<=waiting;
          endcase
          sendingDataPacket<=1'b1;
        end
doReadCmd: begin //send read command
          fifoRead<=1'b0;
          rfifoRead<=1'b0;
          rfifoWrite<=1'b0;
          rfifoClr<=1'b1;
          txData<=(state==moveData) ? etherIData : txData;
          txStart<=(state==sendData);
          etherIAddr<=(state==moveData) ? etherIAddr+1'b1 :
              etherIAddr;
          cmd<=(state==waiting && etherIAddr>readEnd) ?
              doReadData : cmd;
          byteCount<=8'b0;
          waitCount<=20'b0;
          sendingDataPacket<=1'b1;
          case(state)
            waiting:   state<=(txBusy==1'b0 && etherIAddr<=
                readEnd) ? moveData : waiting;
            moveData: state<=sendData;
```

```
            sendData:    state<=(txBusy==1'b1) ? waiting :
                sendData;
            default:   state<=waiting;
        endcase
        commandReceived<=1'b0;
        command<=command_null;
    end
doReadData: begin //read data
        fifoRead<=1'b0;
        rfifoRead<=1'b0;
        rfifoClr<=1'b0;
        etherIAddr<=preambleStart;
        waitCount<=(waitCount==20'hfffff ? waitCount :
            waitCount+1'b1);
        cmd<=((waitCount>20'hf0000 || byteCount==8'hff)
            && state==rWaiting) ? doPreamble : doReadData
            ;
        sendingDataPacket<=(state==rWriteData && ((
            byteCount==8'h03 && rfifoIn!=8'h30 && rfifoIn
            !=8'h4f && justSentStatus==1'b0) || (
            byteCount==8'h04 && rfifoIn!=8'h0D && rfifoIn
            !=8'h4b && justSentStatus==1'b0)) ? 1'b0 :
            sendingDataPacket); //checks if the third
            received character is '0', and fourth is <CR
            >, if so, send a data packet; todo: do this
```

```
            better
        commandReceived<=(state==rWriteData && ((
            byteCount==8'h05 || byteCount==8'h06))) ? ((
            byteCount==8'h05) ? (rfifoIn==command_start)
            : ((byteCount==8'h06 && rfifoIn!=
            command_start) ? 1'b0 :commandReceived)) :
            commandReceived;
        command<=(state==rWriteData && commandReceived
            ==1'b1 && ((byteCount==8'h07 || byteCount==8'
            h08))) ? ((byteCount==8'h07) ? rfifoIn : ((
            byteCount==8'h08 && rfifoIn!=command) ?
            command_null : command)) : command;
        rfifoIn<=(state==rMoveData) ? rxData : rfifoIn;
        rfifoWrite<=(state==rWriteData);
        byteCount<=(state==rMoveData) ? byteCount+1'b1 :
            byteCount;
        case(state)
          rWaiting:    state<=(rxDataReady==1'b1) ?
            rMoveData : rWaiting;
          rMoveData:   state<=rWriteData;
          rWriteData:  state<=rWaiting;
          default:   state<=rWaiting;
        endcase
      end
doPreamble: begin //send preamble
```

```
fifoRead <=1'b0;
rfifoRead <=1'b0;
rfifoWrite <=1'b0;
byteCount<=8'b0;
waitCount<=20'b0;
if((fifoUsedWords>maxByteCount) || (
   sendingDataPacket==1'b0)) begin
  txData<=(state==moveData) ? etherIData :
     txData;
  txStart<=(state==sendData);
  etherIAddr<=(state==moveData) ? etherIAddr+1'
     b1 : etherIAddr;
  cmd<=(state==waiting && etherIAddr>preambleEnd
     ) ? doSendData : cmd;
  case(state)
    waiting:  state<=(txBusy==1'b0 && etherIAddr
      <=preambleEnd) ? moveData : waiting;
    moveData: state<=sendData;
    sendData:   state<=(txBusy==1'b1) ? waiting
      : sendData;
    default:  state<=waiting;
  endcase
end
case(command)
  command_data_prep: sendRaw<=1'b0;
```

```
            command_data_raw: sendRaw<=1'b1;
            default: sendRaw<=sendRaw; //nop
          endcase
        end
doSendData: begin //transmit data
          etherIAddr<=preambleStart;//readStart;
          rfifoWrite<=1'b0;
          if(sendingDataPacket==1'b1) begin //sending
             regular data packet
            fifoRead<=(state==readRq && (byteCount<
               maxByteCount-1'b1)); //only read during
               readRq
            rfifoRead<=1'b0;
            txData<=(state==moveData) ? ((byteCount==
               maxByteCount) ? 8'h0D : ((byteCount==
               maxByteCount-1'b1) ? header_data : fifoData
              )) : txData;
            justSentStatus<=1'b0;
          end
          else begin //sending other packet
            fifoRead<=1'b0;
            rfifoRead<=((state==readRq) && (byteCount<
               maxByteCount-1'b1) && (rfifoEmpty==1'b0));
               //only read during readRq
```

```
  txData<=(state==moveData) ? ((byteCount==
     maxByteCount) ? 8'h0D : ((byteCount==
     maxByteCount-1'b1) ? header_debug : ((
     rfifoEmpty==1'b1) ? 8'h0 : rfifoOut)) ) :
     txData;
  justSentStatus<=1'b1;
end

txStart<=(state==sendData);
waitCount<=(waitCount==20'hfffff ? waitCount :
   waitCount+1'b1);
byteCount<=(state==moveData) ? byteCount+1'b1 :
   byteCount;
//cmd<=(state==waiting && byteCount>maxByteCount
   ) ? doReadCmd : cmd;
cmd<=(state==doneData && waitCount>20'hf0000) ?
   doReadCmd : cmd;
case(state)
  waiting:   state<=(byteCount>maxByteCount) ?
     doneData : (((fifoEmpty==1'b0 ||
     sendingDataPacket==1'b0) && txBusy==1'b0) ?
      readRq : waiting);
  readRq:    state<=moveData;
  moveData:  state<=sendData;
```

```
                sendData:    state<=(txBusy==1'b1) ? waiting :
                    sendData;
                doneData:  state<=doneData;
                default:   state<=waiting;
              endcase
            end

        endcase
      end
    end
endmodule
```

Bibliography

[1] S. B. Leeb, "A conjoint pattern recognition approach to nonintrusive load monitoring," Ph. D. dissertation, Dept. Elect. Eng. Comput. Sci. , Mass. Inst. Technol. , Cambridge, MA, Feb. 1993.

[2] W. M. Siebert *Circuits, Signals and Systems*, The MIT Press, 1986.

[3] L. C. Washington, *Introduction to Cyclotomic Fields*, Springer-Verlag, 1997.

[4] S. Lang, *Algebraic Number Theory*, Springer-Verlag, 1994.

[5] J. M. Masley and H. L. Montgomery, "Cyclotomic fields with unique factorization," J. Reine Angew. Math. 286/287, pg. 248-256, 1976.

[6] W. Wichakool, A. Avestruz, R. W. Cox and S. B. Leeb, "Resolving Power Consumption of Variable Power Electronic Loads Using Nonintrusive Monitoring," PESC 2007, pg. 2765-2771, 2007.

[7] H. Cohen, *A Course In Computational Algebraic Number Theory*, Springer-Verlag, 1996.

[8] A. P. Dempster, N. M. Laird, D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," Journal of the Royal Statistical Society, Series B 39:1-38, 1977.

[9] M. Jamshidian, R. I. Jennrich, "Acceleration of the EM Algorithm by using Quasi-Newton Methods," Journal of the Royal Statistical Society, Series B 59:569-587, 1997.

[10] S. B. Leeb, S. R. Shaw, and J. L. Kirtley, "Transient event detection in spectral envelope estimates for nonintrusive load monitoring," IEEE Trans. Power Del. , vol. 7, no. 3, pp. 1200-1210, Jul. 1995.

[11] S. R. Shaw, C. B. Abler, R. F. Lepard, D. Luo, S. B. Leeb, and L. K. Norford, "Instrumentation for high performance nonintrusive electrical load monitoring," Trans. ASME J. Sol. Energy Eng. , vol. 120, no. 3, pp. 224-229, Aug. 1998.

[12] S. R. Shaw, "System identification techniques and modeling for non-intrusive load diagnostics," Ph. D. dissertation, Mass. Inst. , Technol. , Cambridge, MA, Feb. 2000.

[13] "The Smart Grid: An Introduction," U. S. Department of Energy, http://www. oe. energy. gov/1165. htm.

[14] U. S. Patent 5,548,527, "Programmable Electrical Energy Meter Utilizing a Non-Volatile Memory," August 20, 1996.

[15] U. S. Patent 6,615,147, "Revenue Meter with Power Quality Features," September 2, 2003.

[16] U. S. Patent 7,525,423, "Automated Meter Reading Communication System and Method," April 28, 2009.

[17] http://www.p3international.com/products/special/P4400/P4400-CE.html.

[18] F. Itakura, "Minimum Prediction Residual Principle Applied to Speech Recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol ASSP–23, February 1975, pp. 67–72.

[19] R. E. Abbot and S. C. Hadden, EPRI Final Report CU-6623, "Requirements for an Advanced Utility Load Monitoring System, December 1989.

[20] S. R. Shaw, "System identification techniques and modeling for nonintrusive load diagnostics, Ph. D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, Feb. 2000.